UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Mitar Milutinović

Nedeterministično in leno ogrodje za podatkovno pretokovno računanje

Delo je pripravljeno v skladu s Pravilnikom o podeljevanju Prešernovih nagrad študentom, pod mentorstvom prof. dr. Blaža Zupana.

Ljubljana, 2011

Povzetek

V okviru tega dela smo v programskem jeziku Haskell razvili nedeterministično in leno ogrodje *Etage* za podatkovno pretokovno računanje. Programi v razvitem ogrodju so mreže operacij in povezav, po katerih se prenašajo podatki in kjer se operacije prožijo, ko imajo na voljo vse potrebne podatke. Zaporedje izvajanja ukazov ni definirano vnaprej, ampak je nedeterministično, odvisno od vrstnega reda prihoda podatkov. Hkrati je ogrodje leno, kar pomeni, da izračuna le tiste podatke, ki jih resnično potrebuje.

Predstavljeno ogrodje je še posebej primerno za razvoj programskih sistemov, ki so povezani s fizičnim svetom. Primer takega sistema je krmiljenje robota v realnem času. Narava obdelovanja senzoričnih podatkov in krmiljenja robota je namreč podobna podatkovno pretokovnemu računanju in se lahko zato uspešno preslika v program. To tudi preizkusimo in uspešno potrdimo na testnem robotu. V namene vrednotenja poleg ogrodja razvijemo robota, ustrezen sistem za zajem robotovega položaja v poligonu in komunikacijo z robotovim krmilnim sistemom. Raziskava pokaže na uspešnost uporabe razvitega ogrodja in primernost za razvoj tovrstnih sistemov. Ogrodje dodatno, zmogljivostno, ocenimo še na primeru računanja najcenejše poti v grafu. Ugotovimo, da je delovanje programov na osnovi ogrodja počasnejše od uveljavljenih pristopov, a primernejše za vzporedno izvajanje.

Predstavljeno ogrodje je po našem vedenju prvo te vrste. Razvili smo tudi prvi programski vmesnik do robotovega krmilnega sistema v programskem jeziku Haskell. Vsa razvita programska koda je prosto dostopna na spletu. Ob samem ogrodju je pomembni prispevek dela tudi prikaz možnosti uporabe funkcijskih jezikov za razvoj praktičnih podatkovno pretokovnih sistemov.

Ključne besede:

podatkovno pretokovno računanje, ogrodje, vzporedno izvajanje, robot, krmiljenje, grafi, najcenejša pot, tok podatkov, lena evaluacija, Haskell

Abstract

We have developed a nondeterministic and lazy framework for data-flow computation called *Etage*. We implemented it in Haskell programming language. Programs in the developed framework are networks of operations and connections over which data is transmitted and where operations are triggered when all required data is available. The execution order of operations is nondeterministic and depends on the data arrival order. *Etage* is a lazy framework as it computes only the data really used by operations.

The proposed approach is suitable for development of software systems which are connected to the physical world. An example of such a system is real-time robot control. The nature of sensory data processing and robot control is similar to data-flow computing and can be successfully translated into a program. In the thesis we tested such coupling of the physical machine and *Etage* by developing a robot, a system for capturing its position in a testing polygon, and software for communication with its control system. We found that the developed framework is very suitable for such systems. Additionally, we tested the performance of the framework on the problem of cheapest path search in a graph. Our *Etage*-based search programs performed slower than state-of-the-art solutions, but were more suitable for parallelization.

The presented framework is by our knowledge the first one of its kind. We have also developed the first programming interface for robot's control system in Haskell programming language. The developed open-source code is freely accessible on the web. Alongside the framework itself the main contribution is a showcase for utility of functional languages for development of practical data-flow systems.

Keywords:

data-flow computation, framework, parallel evaluation, robot, controling, graphs, shortest path, data flow, lazy evaluation, Haskell

Kazalo

1 Uvod **1**

2 Pregled uporabljenih tehnologij in sorodnih rešitev **6**
2.1 Haskell 7
2.1.1 Sistem tipov 8
2.1.2 Lenost oziroma nestriktnost 11
2.1.3 Vzporedno izvajanje 13
2.2 Sorodne rešitve 15
2.3 Lego Mindstorms NXT 19
2.3.1 Bluetooth 20
2.3.2 NXT-G 21
2.4 Druge uporabljene tehnologije 25
2.4.1 Spletna kamera 25
2.4.2 Linux 25

3 Ogrodje za podatkovno pretokovno računanje **27**
3.1 Smernice razvoja 28
3.2 Arhitektura ogrodja 31
3.2.1 Operacije 31
3.2.2 Podatki 34
3.2.3 Povezave 38
3.2.4 Gradnja mreže 42
3.3 Osnovne operacije 48
3.3.1 Generator zaporedja 48
3.3.2 Izpisovanje 49
3.3.3 Apliciranje funkcije 50
3.3.4 Zakasnitev 51

3.3.5 Delavec 52

4 Primer uporabe: računanje najcenejše poti v grafu 54
4.0.6 Opis delovanja 55
4.0.7 Analiza delovanja 58

5 Primer uporabe: krmiljenje robota 62
5.1 Zasnova robota 63
5.2 Določanje položaja robota 66
5.2.1 Izračun položaja v prostoru glede na sliko 69
5.2.2 Implementacija 76
5.3 Upravljanje z robotom 78
5.3.1 Direktno upravljanje 79
5.3.2 Posredno upravljanje z vmesnim programom 80
5.4 Umerjanje zavijanja 81
5.5 Krmiljenje robota 83
5.6 Vožnja do tarče 86

6 Sklepne ugotovitve 90

Literatura 94

A Program za računanje najcenejše poti v grafu 100

B Program za vožnjo robota do tarče 106

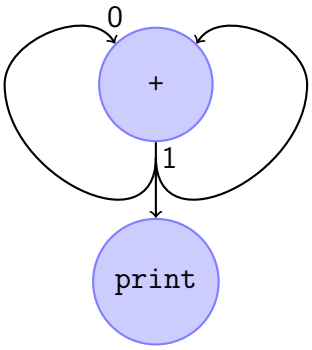
Poglavje 1

Uvod

Današnji računalniki za splošno rabo in njihovi programi večinoma delujejo na podlagi izvajanja zaporedij ukazov, ki delujejo nad podatki. Takšen model *ukazno pretokovnega* (angl. control-flow) računanja se je prijel zato, ker ga je s trenutno tehnologijo možno učinkovito realizirati, hkrati pa je velikemu številu današnjih uporab računalnikov pisan na kožo. Ljudem je navadno tudi lažje razmišljati in sklepati o delovanju programov na podlagi ukazov, ki jih (naj) ti izvajajo zaporedoma. Slednje je morda predvsem stvar navade zaradi prevladovanja tega modela. Primer takšnega programa v obliki izvorne (pseudo)kode je prikazan na sliki 1.1.

```
a := 0
b := 1
repeat:
    print b
    c := b
    b := b + a
    a := c
```

Slika 1.1: Primer ukazno pretokovnega programa, ki izpisuje Fibonaccijevo zaporedje: 1, 1, 2, 3, 5, 8, 13 ...



Slika 1.2: Primer podatkovno pretokovnega programa, ki računa Fibonaccijevo zaporedje: 1, 1, 2, 3, 5, 8, 13 ...

A za določene problemske domene je primernejši drugačen model računanja. Ta temelji na *toku podatkov* (angl. data-flow). V takšnem podatkovno pretokovnem modelu podatki vstopajo v sistem in glede na to, kako se gibljejo (tečejo) med operacijami določajo katere operacije se naj izvedejo. V primeru programov tako ti niso zaporedje ukazov, ampak mreža (graf) usmerjenih povezav, ki določajo, kako naj se podatki prenašajo med operacijami. Operacije pa se omogočijo (izvedejo) takrat, ko imajo na voljo vhodne podatke, in svoj rezultat pošljejo po povezavah naprej do naslednjih operacij. Primer takšnega programa v obliki grafičnega prikaza je prikazan na sliki 1.2.

Program s slike 1.1 vsebuje tri spremenljivke, katerim ukazi zaporedoma in po prvih dveh ukazih v neskončni zanki prirejajo vrednosti. V vsakem obhodu zanke se izpiše trenutno vrednost, zatem pa se še izračuna novo vsoto dveh spremenljivk. Vrstni red ukazov (kot tudi vrstni red prirejanja vrednosti) je nadvse pomemben, saj ravno ta določa delovanje programa.

Program na sliki 1.2 je abstraktna predstavitev podatkovno pretokovnega programa s podanima začetnima vrednostima. Operacije se omogočijo (izvedejo), ko so na voljo vhodni podatki. Tako se začetna vrednost 1 prenese

do operacije `print`, ki jo izpiše, hkrati pa se prenese po dveh povezavah do operacije +. Na prvem vhodu te operacije je že na voljo druga začetna vrednost 0, tako da se tam 1 postavi v vrsto. Drugi vhod se medtem zapolni in omogoči izvedbo operacije. Tako se seštejeta 0 in 1 in vse skupaj se ponovi. Ponovno se izpiše 1, v naslednjem iteraciji 2 in tako dalje.

Vrstni red izvajanja operacij je tako odvisen od toka podatkov po povezavah. Prav tako pa so tudi operacije odvisne od vrednosti, ki so v danem trenutku na njenih vhodih. To izkoriščamo tudi v podanem primeru, saj s pravim zaporedjem prihajanja vrednosti na vhoda operacije + dosežemo, da so vrednosti na prvem vhodu zakasnjene za eno vrednost, kar je ravno to, kar omogoča seštevanje predhodne in nove vrednosti.

Oba programa izpišeta enako zaporedje. In tudi v splošnem je tako, da se v obeh modelih računanja da izračunati vse, kar se izračunati da. Tako je razlika predvsem v tem, kateri model in kateri zapis programa je bolj primeren za konkreten problem oziroma izračun. In še tu je tako, da sta ta dva modela le dve skrajnosti, saj se da oba modela tudi prepletati. Operacija v podatkovno pretokovnem modelu lahko naprimer interno deluje po ukazno pretokovnem modelu.

O teh dveh modelih računanja lahko govorimo tako v okviru strojne [31] kot programske opreme. V okviru tega dela se bomo ukvarjali predvsem s podatkovno pretokovnim računanjem v okviru programske opreme, saj se ta lahko realizira tudi na strojni opremi, ki deluje ukazno pretokovno. Zanimiva lastnost takšnih realizacij je tudi, da jih je možno veliko lažje porazdeliti preko več procesorjev, računalnikov oziroma sistemov, če je to potrebno.

Verjetno najbolj poznan splošen primer podatkovno pretokovnega računanja je računanje v programih za delo s preglednicami, kjer je ponavadi zaželjeno, da se sprememba vrednosti v eni celici takoj pozna v drugih ce-

licah, ki temeljijo na tej celici. Tako sprememba vrednosti podatka v celici povzroči prenos te nove vrednosti v odvisne celice, kjer se definirane operacije ponovno izračunajo in nove vrednosti se prikažejo ter prenesejo naprej.

Drug takšen primer so sistemi za vizualno programiranje, kjer lahko programer določi program tako, da vizualno razporedi funkcijske gradnike in jih ustrezno poveže za prenos podatkov med njimi. Primer takšnega sistema za programiranje je tudi NXT-G za programiranje krmilne enote Lego Mindstorms NXT (več o njej v poglavju 2.3), ki ga bomo bolj podrobno predstavili kasneje v razdelku 2.3.2.

V okviru tega dela bomo raziskali vprašanje, kakšen bi bil splošen in razširljiv sistem za programiranje po podatkovno pretokovnem modelu, ki pa bi hkrati omogočal izrazno moč, kot je je programer navajen pri programiranju po ukazno pretokovnem modelu v obliki izvorne kode. To delo podaja možen odgovor na to vprašanje v obliki nedeterminističnega in lenega ogrodja, ki smo ga poimenovali *Etage*.

Ogrodje omogoča programerju določitev operacij in povezav med njimi, pri čemer poskrbi za njihovo pravilno izvajanje po podatkovno pretokovnem modelu. V ta namen uporabljamo tudi zmogljivosti sistema tipov programskega jezika Haskell, v katerem smo ogrodje implementirali. Sistem tipov uporabimo tudi za modularno zasnovo programov in operacij.

Ogrodje smo preizkusili na dveh primerih uporabe: računanje najcenejše poti v grafu in krmiljenje robota. Računanje najcenejše poti je abstrakten problem, medtem ko smo za krmiljenje robota izdelali le-tega iz Lego kock na osnovi krmilne enote Lego Mindstorms NXT. Prav tako samo razvili tudi zajem robotovega položaja v poligonu z uporabo spletne kamere. Vse to smo povezali v ogrodje in pokazali, kako sta branje senzorjev in komunikacija z robotom lahko enakovredni operaciji ostalim operacijam programa. Slednje

je tudi primer naše bolj splošne ugotovitve, da je računanje po podatkovno pretokovnem modelu veliko bolj primerno za programiranje sistemov, ki so povezani s fizičnim svetom. Primer računanja najcenejše poti pa smo uporabili za ocenitev zmogljivostnih lastnosti našega ogrodja, kjer smo izmerili le za konstantni faktor počasnejše delovanje in po drugi strani boljše delovanje v primeru delovanja na več CPE.

Omenjeni izdelki in rezultati tega dela so ponujeni prosto na spletu pod odprto licenco, kar omogoča drugim njegovo prosto uporabo in nadaljnje delo, razvoj in raziskovanje na njihovi osnovi.

V poglavju 2 predstavimo osnovne gradnike tega ogrodja. Na začetku funkcijski programski jezik Haskell v podpoglavju 2.1. Ker kasneje ogrodje preizkusimo na primeru robota sestavljenega iz Lego kock (na sliki 5.1) na osnovi krmilne enote Lego Mindstorms NXT, jo predstavimo v podpoglavju 2.3. Za tem temeljito predstavimo ogrodje *Etage* v poglavju 3 s primerom iskanja najcenejše poti v grafu v poglavju 4. Ogrodje preizkusimo še na realnem problemu v poglavju 5 na primeru krmiljenja robota. Vse skupaj povzamemo in kritično analiziramo v poglavju 6.

Poglavje 2

Pregled uporabljenih tehnologij in sorodnih rešitev

Ogrodje *Etage*, predstavljeno v tem delu, gradi na ramenih drugih po principu odprte kode: implementirano je v programskem jeziku Haskell, v primeru uporabe pa smo uporabili še knjižnico OpenCV, robota na osnovi Lego Mindstorms NXT in operacijski sistem Linux. Uporaba in delovanje na obstoječih odprtokodnih sistemih ima v splošnem nekaj pozitivnih lastnosti:

- Ker je možen vpogled v izvorno kodo vseh sistemov, se je možno seznaniti s točnim notranjem delovanjem vseh uporabljenih sistemov. To omogoča kvalitetnejšo implementacijo ogrodja, saj omogoča hkrati boljšo široko sliko celotnega prepleta sistema in ogrodja kot tudi razumevanje podrobnosti.
- Če uporabljenemu sistemu manjka kakšna funkcionalnost, ki bi olajšala implementacijo ogrodja, hkrati pa bi bila uporabna tudi drugim uporabnikom, jo je možno razviti v okviru uporabljenega sistema in jo posredovati njegovim razvijalcem, da jo vključijo v uradno različico sistema in s tem omogočijo njeno uporabo tudi drugim.

- V primeru odkritih napak ali nezdružljivosti, se je možno poglobiti v kodo vseh udeleženih sistemov, jih odkriti, odpraviti in popravke posredovati razvijalcem teh sistemov. Tako se problem odpravi tudi za ostale uporabnike teh sistemov, hkrati pa to pomeni tudi pravilno delovanje ogrodja samega.

- Vsi rezultati in koda ogrodja so prav tako na voljo drugim, da si ogledajo implementacijo, se iz nje kaj naučijo, odkrijejo morebitne napake oziroma se domislijo kakšne izboljšave. Tako lahko tudi sami prosto gradijo na tem naprej.

V nadaljevanju se bodo predstavili nekateri uporabljeni odprtokodni sistemi in druge tehnologije, opisana bo njihova vloga v okviru tega dela ter tudi morebitne obstoječe in sorodne rešitve.

2.1 Haskell

Haskell [15] je standardiziran [22] splošno namenski len funkcijski programski jezik. Prav tako je eden tistih, ki so deležni veliko razvoja in raziskovanja[1] in ima tudi veliko skupnost uporabnikov ter razvitih veliko knjižnic[2]. Razširjen je bil z veliko novostmi iz področja teorije programskih jezikov in pogosto služi za referenčno implementacijo novih konceptov iz tega področja.

Funkcijski programski jeziki so družina jezikov, kjer se razume funkcijo v matematičnem smislu in ne v siceršnjem imperativnem smislu. Tako funkcije v splošnem (če so čiste) ne spreminjajo stanja oziroma podatkov in ne interaktirajo z okoljem (bolj splošno: nimajo stranskih učinkov), ampak vrnejo

[1] `http://www.haskell.org/haskellwiki/Research_papers`
[2] `http://hackage.haskell.org/`

izhodno vrednost le glede na vrednosti vhodnih argumentov. Za enake vrednosti tako funkcija vedno vrne enako izhodno vrednost, ne glede na to, kdaj je bila evaluirana. Programi v funkcijskem programskem jeziku evaluirajo izraze za razliko od imperativnih, ki so sestavljeni iz stavkov, ki spreminjajo globalno stanje, ko se izvajajo. V funkcijskih programskih jezikih so funkcije enakovredne drugim vrednostim in se naprimer lahko podajajo kot argumenti drugim funkcijam. Pravimo, da so funkcije *prvorazredne vrednosti* (angl. first-class values) programskega jezika.

2.1.1 Sistem tipov

Haskell uporablja uporablja zelo razvit sistem tipov in s tem posredno zmanjšuje število napak v samem programiranju, saj programski jezik strogo zahteva statično (ob prevajanju) pravilnost tipov. Zaradi tega se lahko z večjo gotovostjo razvijejo stabilni in pravilno delujoči programi oziroma sistemi.

Poglavitna značilnost sistema tipov je podpora *razredom tipov* (angl. type classes) [33], ki omogočajo inovativen način definiranja skupnih funkcij nad različnimi podatkovnimi tipi (naprimer definiranje funkcije enakosti dveh vrednosti) in s tem modularnega razširjana funkcionalnosti jezika nad nove podatkovne tipe. Sčasoma so bile odkrite še druge uporabe razredov tipov. Tako razred, parametriziran s tipom, deklarira metode (funkcije) nad tem tipom, programer pa lahko potem za željene tipe implementira metode tega razreda (definira instanco razreda). Za razliko od objektno usmerjenega programiranja tu govorimo, da so tipi instance razreda tipov. S tem se lahko uporabljajo metode razreda nad vsemi vrednostmi vseh tipov, ki so instance tega razreda. Pomembna lastnost tega pristopa je modularnost, ki omogoča, da se instance razredov nad tipi definirajo neodvisno od samega

razreda tipov. Tako lahko ob definiciji novega tipa določimo, da ta pripada (je instanca) željenim razredom tipov in zaradi tega lahko uporabljamo na vrednostih tega tipa vse funkcije razredov in druge funkcije, ki posredno uporabljajo te funkcije.

Primer kasnejše uporabe razredov tipov so naprimer tudi monade [25], ki so splošno ogrodje in lahko modelirajo različne vrste izračunov. Najbolj pogosto izračune s stranskimi učinki. Haskell je namreč čist funkcijski programski jezik, kjer je tip (monada), ki predstavlja stranske učinke, ortogonalen na tip (čistih) funkcij. Tako se ob preverjanju tipov zagotovi, da deli programa, ki naj ne bi imeli stranskih učinkov, teh res nimajo (sicer preverjanje ujemanja tipov ne uspe). Praktična in koristna lastnost tega pristopa je tudi, da programer lahko že iz tipa funkcije razbere, če ima ta lahko stranske učinke ali ne. Za uporabo monad je v Haskellu na voljo *sintaktični sladkorček* (angl. syntactic sugar), ki omogoča zapis bolj podoben imperativnim programskim jezikom, namreč kot zaporedje ukazov. Imenuje se *do-zapis* (angl. do-notation), saj se začne s ključno besedo do.

Monade se lahko gnezdi. Tako se lahko na osnovno monado s pomočjo transformacij dodajo zaporedoma še ostale [17]. Naprimer na monado IO, ki je osnovna monada za interakcijo z okoljem programa, se lahko s pomočjo $StateT$ transformacije dobi nova monada, ki podpira tudi stanje. S tem se lahko različne monade, ki prispevajo vsaka svojo funkcionalnost, sestavi v željeno kombinacijo, kjer se le-ta odraža tudi v tipu končne monade.

Novejši prispevek k Haskellovemu sistemu tipov so *indeksirane družine tipov* (angl. indexed type families) [26]. Družine tipov so konstruktorji tipov, ki predstavljajo množico tipov. Razlika med navadnimi parametriziranimi konstruktorji tipov in konstruktorji družin je precej podobna razliki med parametrično polimorfičnimi funkcijami in metodami razredov tipov. Pri

razredih tipov definiramo skupne funkcije, katere delujejo nad vrednostmi tipov, ki so instance teh razredov. Pri družinah tipov pa definiramo skupne funkcije nad tipi samimi. Tako pridobimo še en nivo modularnosti.

Funkcijska narava jezika in razredi tipov (ter njihova nadgradnja z družinami tipov) so v okviru tega dela pomembni predvsem kot pristop k modularni oziroma razširljivi zasnovi ogrodja [16]. V primerjavi z bolj poznanimi pristopi k modularnemu programiranju (naprimer iz objektno usmerjenega programiranja) omogočajo modularno razširjanje podatkovnega tipa in funkcij nad podatkovnim tipom, medtem ko ohranjajo statično pravilnost tipov. V primeru našega ogrodja to pomeni, da je možno razširjanje tipa podatkov, ki potujejo po povezavah, in funkcij nad njim skozi dodatne module, ki jih lahko naprimer prispeva uporabnik ogrodja ali kakšna knjižnica, ki ogrodje razširja na bolj specifično problemsko domeno.

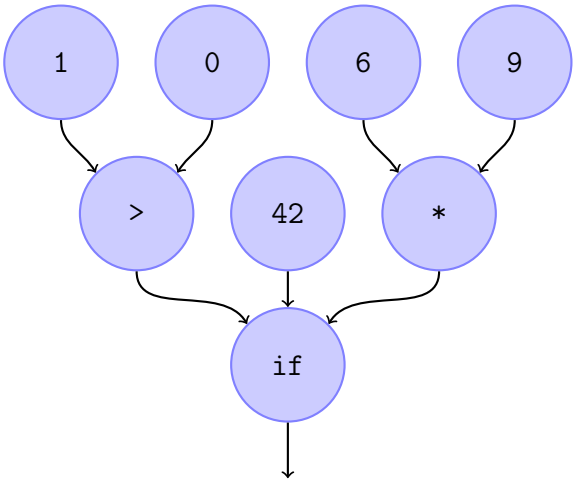
Splošno ime za takšen način modularnosti je *problem izraza* (angl. expression problem) [32]. Predstavljamo si ga lahko na primeru knjižnice za računanje aritmetičnih izrazov, ki naprimer podpira operaciji seštevanja in odštevanja nad naravnimi števili. Knjižnico bi želeli z dodatnim (neodvisno prevedenim) modulom razširili tako, da bi podpirala še množenje in deljenje in po novem vse štiri operacije tudi nad števili s plavajočo vejico, medtem ko bi se ohranjala statična pravilnost tipov. V objektno usmerjenih programskih jezikih se ponavadi v ta namen uporabi *vzorec obiskovalca* (angl. visitor pattern), ki pa ne zadovolji pogoja glede statične pravilnost tipov, ali kakšno od bolj kompleksnih razširitev, kot so naprimer mixini. V Haskellu se ta problem lahko elegantno reši kar z razredi tipov, če se zapiše tipe podatkov in funkcije v odprti obliki [18].

2.1.2 Lenost oziroma nestriktnost

Lastnost čistih funkcij, da vedno vrnejo enako vrednost za enake vhodne podatke se imenuje *referenčna transparentnost* (angl. referential transparency), kar pomeni, da je vseeno, kdaj se te funkcije evaluirajo. Zato je možno odložiti evaluacijo le-teh prav do trenutka, ko je rezultat funkcije potreben. Temu se reče *lena evaluacija* (angl. lazy evaluation), izogiba se nepotrebnemu računanju in omogoča definicijo in uporabo neskončnih podatkovnih struktur. Zaradi tega se Haskell označuje kot len funkcijski programski jezik (čeprav v resnici je definiran le kot (po privzetem) nestrikten). Podpira pa označitev konkretnih podatkovnih tipov in aplikacij funkcij kot striktnih, če programer oceni, da je to koristno zaradi zmogljivostnih razlogov ali za zmanjšanje porabe spomina.

Lena evaluacija je tehnično kombinacija nestriktne (denotacijske) semantike in deljenja vrednosti (ista vrednost se lahko uporabi večkrat na različnih koncih programa in se ne evaluira vsakič znova). Nasprotje je *neučakana evaluacija* (angl. eager evaluation), ki je implementacija striktne (denotacijske) semantike. Nestriktna semantika pomeni, da ima lahko funkcija definirano vrednost, čeprav je kateri njen argument nedefiniran. Nestriktna semnatika omogoča, da se formalno obdeluje neskončne količine podatkov, naprimer v obliki neskončnih podatkovnih struktur. Seveda pa se v praksi obdela le končna množica teh podatkov. Po potrebi je mogoče v Haskellu zahtevati striktno semantiko za evaluacijo konkretnega izraza, hkrati pa tudi v imperativnih programskih jezikih (kjer sicer prevladuje striktna semantika) obstajajo stavki, ki se deloma izvajajo nestriktno. Primer takšnega je naprimer pogojni stavek, kjer se izvede le ustrezna veja stavka in ne obe.

Če si predstavimo izraz v obliki grafa, ki povezuje vhode in izhode funkcij med seboj, lahko z njegovo pomočjo lažje razumemo razliko med striktno in



Slika 2.1: Izraz `if 1 > 0 then 42 else 6 * 9` v obliki grafa.

nestriktno semantiko. Primer takšnega grafa je na sliki 2.1.

Striktna semantika pomeni, da se redukcija (evaluacija) izraza dogaja od znotraj navzven, torej se najprej reducirajo argumenti funkcije in zatem funkcija sama. Na prikazanem grafu primera 2.1 to pomeni redukcijsko od zgoraj navzdol. Nestriktna semantika pa deluje obratno, od zunaj navznoter oziroma na prikazanem grafu od spodaj navzgor. To pomeni, da se najprej reducira `if`, potem > in tako naprej. Razlika je pomembna, saj bo v primeru, da se kakšen (pod)izraz evaluira v napako ali neskončno zanko, kar se v denotacijski semantiki imenuje *dno* (angl. bottom), redukcija od znotraj vedno našla dno, ki se bo propagiralo navzven. V primeru redukcije od zunaj pa lahko določene podizraze eliminirajo zunanje redukcije in se tako lahko le-ti nikoli ne evaluirajo v dno. Tako lahko pri nestriktni semantiki graf izraza vsebuje cikle (dno) in je možno (seveda pa ne nujno), da se celoten izraz uspešno evaluira v vrednost, medtem ko pri striktni semantiki graf izraza ne sme vsebovati cikla (dno), oziroma se, če cikel vsebuje, takšen izraz vedno evaluira v dno (se naprimer evaluacija nikoli ne zaključi).

Zanimivo je, da lahko podobno razlikovanje najdemo tudi v podatkovno pretokovnem računanju [31, stran 56], kjer se analogija leni evaluaciji imenuje *izvajanje po potrebi* (angl. demand-driven execution), kjer se operacijo omogoči (izvede) takrat, ko je potreba za njenim rezultatom. To je različno od siceršnjega pristopa, kjer se operacije omogoči takrat, ko so na vhodih potrebni vhodni podatki, kar se imenuje *izvajanje po podatkih* (angl. data-driven execution). V [9] se najde še drugo poimenovanje: *vlečno osnovana evaluacija* (angl. pull-based evaluation) za izvajanje po potrebi in *potisno osnovana evaluacija* (angl. push-based evaluation) za izvajanje po podatkih.

Analogijo lahko peljemo še naprej in evaluacijo čistih funkcij (izrazov) vidimo kot obliko podatkovno pretokovnega računanja, oziroma redukcije ustreznega grafa povezav in operacij v (izhodno) vrednost. Leno evaluacijo pa kot razširitev na grafe, ki lahko vsebujejo cikle. Tako Haskell na nek način že sam po sebi podpira podatkovno pretokovno računanje in to v splošnejši obliki z dovoljenimi cikli.

2.1.3 Vzporedno izvajanje

Haskell zaradi svojih lastnosti omogoča tudi enostavno in robustno vzporedno računanje ter dobro abstrakcijo le-tega. Ker so čiste funkcije referenčno transparentne, torej nimajo stranskih učinkov, možno jih je evaluirati kadarkoli in vedno vrnejo enako izhodno vrednost za enake argumente, jih je možno izvajati na poljubnem procesorju računalnika v poljubnem vrstnem redu in tudi špekulativno vnaprej. Tako Haskell omogoča enostavno označitev evaluacije vrednosti s strategijami (špekulativnega) vzporednega računanja [19,29] kar z že obstoječo sintakso jezika. Prevajalniku lahko naprimer enostavno namignemo, da bomo potrebovali vse vrednosti iz seznama in da

jih lahko poskusi, če ima na voljo proste procesorje, evaluirati špekulativno in vzporedno vnaprej.

Tu se tudi vidi razlika, zakaj je Haskell v resnici le nestrikten jezik (čeprav se ga pogosto označuje kot lenega), medtem ko je lena evaluacija le eden izmed načinov implementacije nestriktnega jezika (njegova operacijska semantika), ki se trenutno sicer res uporablja v (verjetno) vseh njegovih prevajalnikih, ampak se lahko s strategijami za vzporedno računanje doseže špekulativno evaluacijo, kar več ni lena evaluacija, medtem ko se še vedno ohrani nestriktna semantika.

Ob tem pa so na voljo tudi drugi pristopi k vzporednemu računanju. Eden teh je podpora lahkim nitim, ki omogočajo vzporedno izvajanje funkcij v monadi IO, in asinhronem kanalu za komunikacijo med njimi [23]. Haskell sam skrbi za pravilno delegacijo in razvrščanje teh lahkih niti v izvajanje nitim operacijskega sistema. Na voljo je tudi nadgradnja mehanizma za komunikacijo med temi lahkimi nitmi s *sestavljivimi pomnilniškimi transakcijami* (angl. composable memory transactions) [8, 12], ki omogočajo enostavnejše sestavljanje atomarnih operacij v bolj kompleksne operacije, kjer se spremembe spremenljivk izvajajo znotraj (po potrebi gnezdenih in sestavljenih) transakcij. Tako programerju ni potrebno skrbeti za zaklepanje kritičnih sekcij in s tem tudi ne za smrtne objeme (angl. deadlock) in druge nezaželjene pojave pri njihovi uporabi.

Zelo uporabno je tudi, da vzporedno računanje omogoča tudi v primeru uporabe funkcij oziroma knjižnic napisanih v drugih programskih jezikih, kar naprimer omogoča uporabo visokozmogljivih optimiziranih knjižnic napisanih v programskem jeziku C ali celo zbirniku, medtem ko se Haskell uporabi za visokonivojsko delovanje programa. To smo uporabili za odzivno pridobivanje položaja robota, kar bomo bolj podrobno predstavili v razdelku 5.2.1.

Vse te lastnosti programskega jezika Haskell odlično sovpadajo s problemsko domeno tega dela. Sistem tipov omogoča boljšo modularnost, razširljivost in stabilnost delovanja ogrodja. Dobra podpora vzporednemu računanju zagotavlja, da se lahko bolj kompleksni programi na osnovi ogrodja avtomatično vzporedno računajo. Že omenjena lenost programskega jezika pa omogoča računanje le tistega dela mreže (grafa) programa na osnovi ogrodja, ki je potreben za podane vhodne podatke, in ne vedno znova (za katerikoli vhodni podatek) celotnega. Še več, lenost omogoča tudi boljšo sestavljivost in ponovno uporabo komponent mreže, kjer se lahko osnovna mreža sestavi iz že vnaprej sestavljenih oziroma pripravljenih podmrež, kjer pa se zaradi lenosti na koncu uporabi le potreben/željen del teh podmrež.

V okviru tega dela smo uporabili The Glasgow Haskell Compiler[3] (GHC) [24] prevajalnik za programski jezik Haskell, saj ima implementirane potrebne novosti (oziroma razširitve) jezika kot tudi knjižnice na njihovi osnovi. Uporabili smo razvojno različico 7.1.

2.2 Sorodne rešitve

Zaradi vseh lastnosti Haskella, ki so tudi nas motivirale, da ga uporabimo za implementacijo ogrodja *Etage*, se v njem že najde implementiranih precej raznolikih idej in konceptov, ki bolj ali manj predstavljajo neko obliko podatkovno pretokovnega delovanja.

Eden takšnih je naprimer *model igralca* (angl. actor model) [13], kjer se med igralci (osnovne enote, ki izvajajo operacije) izmenjujejo sporočila, vsak igralec pa se potem na podlagi sporočil lokalno odloča, kako bo deloval, kaj bo izvedel in če bo potrebi ustvaril dodatne igralce oziroma poslal nova sporočila

[3] `http://www.haskell.org/ghc/`

drugim igralcem. Ta model je osnovni način delovanja programskega jezika Erlang [2], za Haskell pa je na voljo v obliki knjižnice[4], ki ga implementira v razširjeni obliki [28], ki omogoča bolj kompleksne načine branja sporočil iz vrste čakajočih.

Če ta model primerjamo z modelom predstavljenim v tem delu, lahko igralce enačimo z operacijami, razlika pa je pri povezavah med njimi. V našem modelu so povezave med operacijami eksplicitne, medtem ko se v primeru modela igralca pojavljajo implicitne povezave glede na to, kako igralci pošiljajo sporočila drugim igralcem. Eksplicitne povezave omogočajo lažje sklepanje o programu, medtem ko implicitne povezave omogočajo bolj dinamičen pristop in se lahko naslovnik sporočila določi med samim delovanjem programa.

Tudi z eksplicitnimi povezavami našega modela je takšno odločanje možno, ampak moramo prvo ustrezno povezati operacijo z naslovnikom oziroma naslovniki. Ustvarjanje novih operacij in povezovanje le-teh med seboj lahko naredimo kar med delovanjem programa, saj iz stališča ogrodja ni razlike kdaj se znotraj delovanja programa kakšna operacija ustvari in poveže. V praksi pa zaradi lažjega razumevanja vse operacije med seboj sestavimo na začetku delovanja programa, kar ogrodje tudi spodbuja s konstrukti, ki jih daje na voljo.

Eleganten pristop in najbolj v duhu funkcijskega programiranja se imenuje *funkcijsko reakcijsko programiranje* (angl. functional reactive programming) [9, 14]. Ta med prvorazredne vrednosti Haskella doda dinamične vrednosti (vrednosti, ki spreminjajo svojo vrednost skozi čas), ki se imenujemo *vedenja* (angl. behaviors), in *dogodke* (angl. events), zaporedja vrednosti s časom. Lahko se jih definira, sestavlja, podaja kot argumente

[4] `http://hackage.haskell.org/package/actor`

funkcijam in iz njih vrača. Vedenja so sestavljena iz nekaj osnovnih primitivnih elementov, kot naprimer konstantnih (statičnih) vedenj in časa (kot ure), s pomočjo zaporednega in vzporednega kombiniranja. Izračuni, ki vsebujejo vedenja, so tako tudi sami vedenja (vračajo spreminjajočo se vrednost skozi čas). Tako se trenutna vrednost vedenja, ki je posredno ali neposredno odvisna od vrednosti drugih vedenj, avtomatično tudi sama spremeni ob spremembi katerikoli vrednosti odvisnih vedenj. Vedenja in dogodki se povežejo preko *preklopnika* (angl. switcher), kjer vsak dogodek definira nov način delovanje vedenja. S tem se lahko načini delovanja vedenj spreminjajo skozi čas.

Ta pristop tako deluje kot vmesna pot med funkcijskimi jeziki, kjer so vrednosti nespremenljive, oziroma referenčno transparentne, in jeziki, kjer se vrednosti spreminjajo, a zato lahko takšna sprememba predstavlja stranski učinek. V funkcijskem reakcijskem programiranju se deklerativno (in nespremenljivo) definirajo odvisnosti med vrednostmi, katerih vrednosti se sicer kasneje spreminjajo, ampak odvisnosti med njimi ostanejo. Na takšen način se lahko še naprej uporablja veliko lastnosti funkcijskih jezikov. Prav tako takšen pristop predstavlja alternativen način interakcije z okoljem programa, saj se lahko vhodi in izhodi iz programa opišejo kot vedenja in so tako enakovredni drugim vedenjem v programu.

Funkcijsko reakcijsko programiranje je aktivno raziskovano in uporabljeno na različnih področjih od računalniške animacije [10], uporabniških vmesnikov [5, 6], do robotike [14, 20, 21] in še drugje.

Če si predstavimo vedenja in odvisnosti (funkcije, s katerimi jih sestavljamo) med njimi, ki jih definiramo v programu, v obliki grafa, lahko vidimo, da ta predstavlja dualen graf grafu, ki ga dobimo pri modelu podatkovno pretokovnega računanja, saj so vozlišča v tem primeru spremenljive vrednosti,

vedenja, oziroma podatki, povezave med njimi pa odvisnosti, funkcije, oziroma operacije nad njimi.

Na podlagi te dualnosti lahko vidimo podobnost med funkcijskim reakcijskim programiranjem in podatkovno pretokovnim računanjem in če ga primerjamo z ogrodjem predstavljenim v tem delu, vidimo, da funkcijsko reakcijsko programiranje predstavlja predvsem bolj eleganten, bolje (denotacijsko) definiran in funkcijski vmesnik nad podobno osnovo, ki s tem pripomore k boljši abstrakciji, boljši sestavljivosti in razvoju kompleksnejših konceptov, ki se jih sicer ne bi opazilo oziroma odkrilo.

Na naše ogrodje lahko zato gledamo kot na enega izmed pristopov k implementaciji koncepta funkcijskega reakcijskega programiranja, brez nadgradnje v obliki elegantne funkcijske abstrakcije, ki namesto nje uporablja predstavitev v obliki eksplicitnih operacij in (podatkovnih) povezav. Hkrati pa tudi naše ogrodje ohranja interakcijo z okoljem programa enakovredno drugim virom podatkov in s tem omogoča enostavno zlitje tako interakcije z uporabnikom, branje vhodnih podatkov, prikaz podatkov in povezav z obstoječimi sistemi in ukazno pretokovnimi knjižnicami, medtem ko se lahko operacije nad njimi definirajo kot čiste funkcije.

Zanimiv je tudi Intelov prispevek na tem področju s podporo Haskellu za svoj Intel Concurrent Collections (CnC)[5]. Intel opisuje CnC kot knjižnico za vzporedno računanje, ki se uporablja za izračune na osnovi grafov, ki si delijo nespremenljive podatke in tabele. Operacije poimenujejo *koraki* (angl. steps), za katere ugotavljajo, da so čiste funkcije, ki preberejo podatke in dajo na podlagi njih rezultat, ki omogoči naslednje korake. Terminologija je tako drugačna, ampak je knjižnica sicer zelo podobna podatkovno pretokovnemu računanju. Z njo želijo izpostaviti prednosti, ki jih imajo tudi v

[5] `http://hackage.haskell.org/package/haskell-cnc`

tem delu izpostavljeni koncepti funkcijskih jezikov, kot so med drugim čiste funkcije, referenčna transparentnost, nespremenljivost podatkov ... pri omogočanju boljše vzporedno izvajanje programov, kar se kaže kot eden temeljnih pristopov k izboljševanju zmogljivostnih lastnostni programov v prihodnosti.

V primerjavi z našim pristopom je CnC omejen le na čiste funkcije za operacije nad podatki, kar posledično poveča možnosti vzporednega izvajanja programa.

Če pogledamo sorodne rešitve zunaj Haskella, je verjetno najbolj podoben našemu ogrodju programski jezik Erlang[6], ki že sam po sebi podpira delovanje po modelu igralca v obliki njegovih procesov in izmenjevanja sporočil med njimi. Ob tem se ga lahko izvaja široko razpršeno in z visoko razpoložljivostjo, pri čemer sam skrbi za vso potrebno režijo. Je stabilen, ima širok krog uporabnikov in uporablja se ga tudi v velikih poslovnih sistemih, kot so naprimer sistemi teleoperaterjev.

Za razliko od našega ogrodja ni len in predvsem ne podpira statične pravilnosti tipov. Lenost omogoča, da program posreduje le informacijo o obstoju podatka, medtem ko se podatek sam ne izračuna, dokler ni potreben. Preverjanje tipov ob prevajanju in še posebej naša nadgradnja s preverjanjem pravilnosti mreže s pomočjo tipov pa omogoča stabilnejše, pravilnejše in predvsem predvidljivejše delovanje.

2.3 Lego Mindstorms NXT

Ogrodje *Etage* bomo v okviru tega dela preizkusili na primeru robota zgrajenega iz Lego kock na osnovi kompleta Lego Mindstorms NXT[7]. Gre za komplet proizvajalca igrač Lego, ki omogoča uporabo Lego kock, predvsem

[6] `http://www.erlang.org/`
[7] `http://mindstorms.lego.com/`

iz serije Lego Technic, za izgradnjo poljubnega robota. Komplet vsebuje krmilno enoto, nekaj osnovnih senzorjev in 3 servo motorje, ki imajo vgrajene tudi optične enkoderje položaja s stopinjsko natančnostjo, tako da lahko služijo tudi kot senzorji oziroma lahko posredujejo povratno informacijo o položaju motorjev.

Krmilna enota lahko izvaja programe naložene nanjo, lahko pa tudi sprejema ukaze preko USB in povezave Bluetooth. Njen operacijski sistem je odprtokoden, prav tako pa so prosto dostopne vse potrebne specifikacije delovanja in uporabljenih protokolov, kar je omogočilo razvoj alternativnih operacijskih sistemov, alternativnih programskih jezikov za programiranje programov, dodatnih senzorjev, kot tudi vmesnikov za oddaljeno upravljanje v najrazličnejših programskih jezikih.

Zaradi vseh naštetih lastnosti, predvsem odprtosti programske opreme in relativno kvalitetnih motorjev v kombinaciji z vsemi konstrukcijskimi možnostmi Lego kock, je tako komplet odlična platforma za razvoj in učenje osnov robotike.

2.3.1 Bluetooth

Uporabna lastnost krmilne enote je tudi podpora brezžični komunikaciji Bluetooth. Ta omogoča brezžično povezavo več krmilnih enot med seboj, za nas pa je zanimiva povezava krmilne enote z računalnikom.

V splošnem je krmilna enota precej omejena s svojo računsko močjo in pomnilnikom (Atmel 32-bit ARM 48 MHz procesor, AT91SAM7S256, 256 KB flash, 64 KB RAM) in ni zadosti zmogljiva za direktno izvajanje kompleksnejših programov, ki naprimer vsebujejo algoritme učenja in pomnjenja naučenih podatkov. Sicer obstaja širok nabor prilagojenih operacijskih sistemov in programskih jezikov namenjenih razvoju programov za izvajanje na

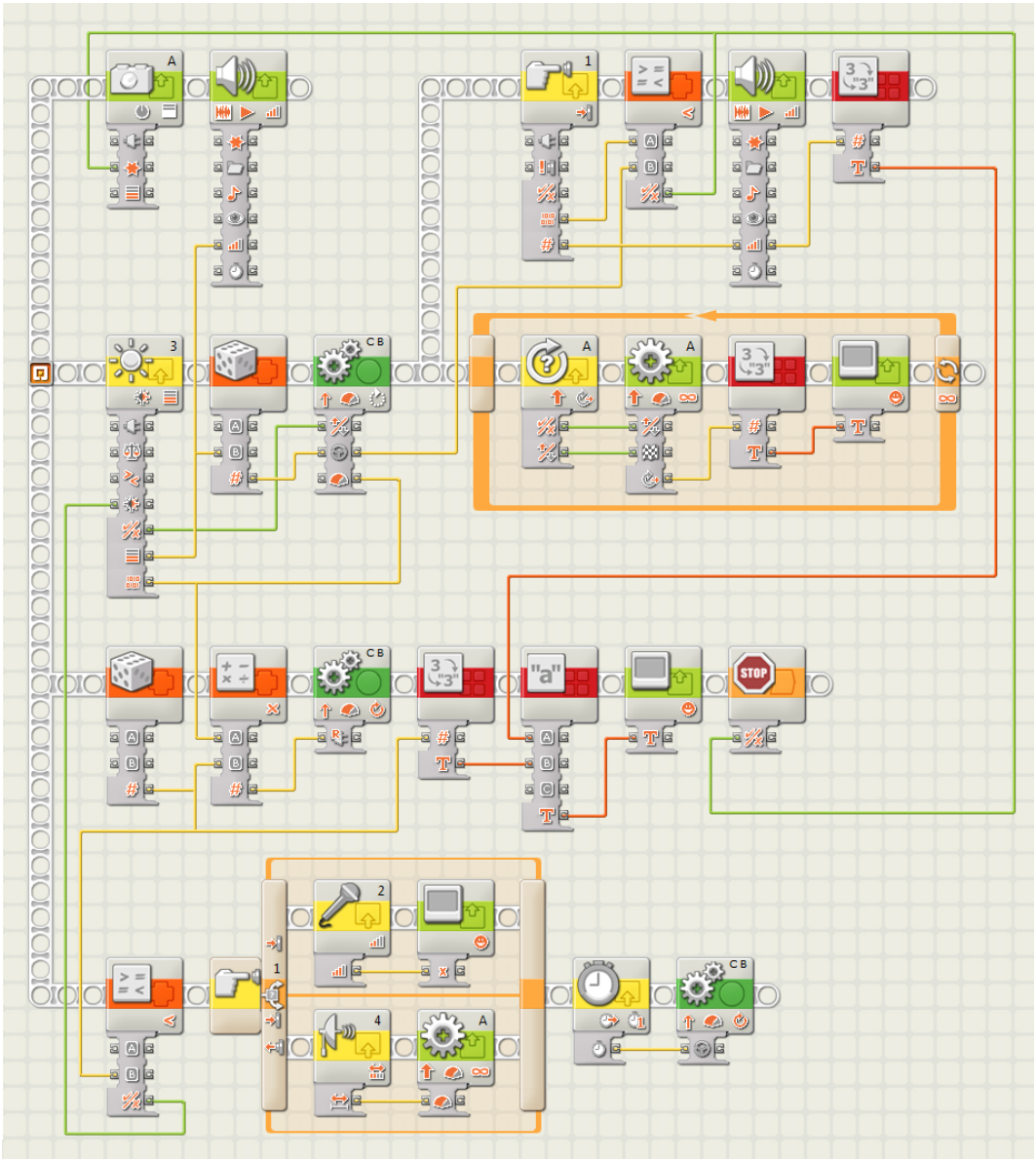
krmilni enoti, ki pa so še vedno precej omejeni, nudijo le osnovne funkcionalnosti, in za današnje čase nizkonivojski, kar onemogoča morebitno uporabo že razvitih knjižnic. V teh primerih je koristna možnost brezžičnega oddaljenega upravljanja krmilne enote iz bolj zmogljivega računalnika.

Na podoben način bomo uporabili komunikacijo Bluetooth tudi v našem primeru, kjer bo v Haskellu razvit program na osnovi *Etage* ogrodja preko povezave Bluetooth krmilil robota, sam pa bo tekel na računalniku. Prednosti in slabosti tega pristopa bomo analizirali v razdelku 5.3.

2.3.2 NXT-G

V kompletu je priložen tudi sistem za programiranje krmilne enote NXT-G. NXT-G je sistem za vizualno programiranje na osnovi vizualnega programskega jezika G in okolja LabVIEW proizvajalca National Instruments. Z njim deli veliko lastnosti, predvsem pa svojo podatkovno pretokovno naravo. Medtem ko je LabVIEW namenjen profesionalnim uporabnikom za hiter razvoj meritvenih in kontrolnih sistemov ter analizo podatkov, za katerega obstaja v obliki dodatka tudi podpora za Lego Mindstorms NXT, je NXT-G poenostavljen in namenski sistem namenjen predvsem začetnikom in le programiranju Lego Mindstorms NXT skozi grafično privlačen in v Lego stilu oblikovan vmesnik.

Vizualni program (na sliki 2.2 primer za sistem NXT-G) je sestavljen iz povezav in z njimi povezanih gradnikov, ki predstavljajo osnovne operacije krmilne enote, od računanja s podatki, preko kontrolnih struktur kot so zanke in večopcijska stikala, do krmiljenja motorjev in branja podatkov iz senzorjev. Ker program deluje podatkovno pretokovno je vrstni red izvajanja gradnikov določen z vrstnim redom zadovoljevanja vhodnih podatkov gradnikov, zato lahko, če želimo točno določen vrstni red, dodamo navidezno



Slika 2.2: Primer vizualnega programa v sistemu NXT-G.

odvisnost med gradniki na podatek (žeton), ki ga sicer v samem gradniku ne uporabimo, določa pa s svojo potjo vrstni red izvajanja. Da bi bili programi bolj pregledni in enostavnejši za začetnike so na voljo že omenjene kontrolne strukture, ki to zakrijejo. Tako lahko programer sestavi gradnike v željenem vrstnem redu, z zankami in ostalimi kontrolnimi strukturami, sistem sam pa poskrbi za ustrezno izvajanje.

Vizualni program je možno uporabiti tudi kot gradnik znotraj drugega programa. Na takšen način je realizirana podpora za modularen razvoj in ponovno uporabo, hkrati pa to naredi kompleksne programe preglednejše, saj zmanjša vizualno velikost končnega programa.

Ko vizualni program dokončamo, ga prevedemo v kodo namenjeno krmilni enoti, prekopiramo nanjo ter ga na njej poženemo. Krmilna enota izvaja program v navideznem okolju in tudi koda je koda tega okolja in ne direktno strojna koda. Sicer ima navidezno okolje in koda zanj nekatere lastnosti, ki delajo prevajanje iz podatkovno pretokovnega programa zanj lažje, vseeno gre za ukazno pretokovno delovanje, kar je razumljivo, saj je tudi sam procesor krmilne enote klasičen ukazno pretokovni procesor. Podoben pristop bomo izbrali tudi mi za naše ogrodje, saj se bo programe navkljub njihovi podatkovno pretokovni naravi izvajalo na splošnem računalniku. Razlika bo, da jih ne bomo prevajali za krmilno enoto, ampak jih bomo izvajali oddaljeno.

NXT-G je tako odličen za začetnike, saj si lahko lažje vizualno predstavljajo tako potek programa (s pomočjo kontrolnih struktur) kot potek podatkov. Gradniki so tudi dovolj visokonivojski, naprimer en gradnik za vse povezano z upravljanjem motorja, da je možno že z nekaj gradniki doseči premikanje robota na podlagi podatkov iz senzorjev, kar pomeni hiter začetek programiranja in položno učno pot s hitrimi rezultati.

Čeprav je v primeru NXT-G programer omejen s podanim naborom gradnikov in njihovimi funkcionalnosti, so le-te razširljive z dodatnimi gradniki, ki se lahko razvijejo v okolju LabVIEW, in kar nekaj jih je prosto na voljo na spletu. Tako se po eni strani ohranja enostavnost sistema NXT-G, po drugi strani pa se omogoča njegovo razširljivost, čeprav je verjetno v primeru potrebe po večjem številu (predvsem lastnih) dodatnih gradnikov boljše kar celoten program razvijati v okolju LabVIEW.

NXT-G prispeva k temu, da se programiranje robotov po podatkovno pretokovnem modelu dozdeva smiselno in izvedljivo. V splošnem si namreč robota lahko predstavljamo kot sistem, v katerega vstopajo podatki iz senzorjev in drugih naprav priključenih nanj, izstopajo pa ukazi za krmiljenje njegovih motorjev in s tem robota samega, kar nakazuje na podatkovno naravo takšnega sistema. To idejo bomo preizkusili tudi na primeru našega ogrodja v poglavju 5.

Če primerjamo NXT-G (in tudi G/LabVIEW) z našim ogrodjem, vidimo da je poglavitna razlika v tem, da se naše ogrodje programira v obliki izvorne kode za razliko od vizualne manipulacije z gradniki in povezavami. Oblika izvorne kode približa programiranje po podatkovno pretokovnem modelu programerjem, ki že imajo izkušnje s programiranjem, hkrati pa ogrodje omogoča tudi uporabo že obstoječih knjižnic (napisanih tudi v drugih programskih jezikih), kot tudi programiranje notranjega delovanja gradnikov po modelu toka ukazov, kar pride prav v primeru implementacije znanih algoritmov, ki so večinoma vsi razviti znotraj tega modela. Medtem ko v NXT-G za povratno informacijo o tem, ali je program pravilno sestavljen, služi predvsem vizualna predstava odnosa med gradniki in povezavami, ki naprimer z barvami označuje povezave različnih podatkovnih tipov, v primeru ogrodja *Etage* za to poskrbi Haskellov sistem tipov ob prevajanju.

Medtem ko je NXT-G namenjen predvsem začetnikom, profesionalen G/LabVIEW pa za hiter razvoj razvoj meritvenih in kontrolnih sistemov, želimo, da je naše ogrodje veliko bolj splošno po uporabi. Želimo si, da služi za povezovanje obstoječih sistemov z novimi in to s pomočjo podatkovno pretokovnega modela, ter da tako predstavlja most med obema modeloma za bolj zahtevne uporabnike in uporabe.

2.4 Druge uporabljene tehnologije

2.4.1 Spletna kamera

Na primeru uporabe našega ogrodja za krmiljenje robota, bomo za glavni in edini senzor uporabili spletno kamero Logitech HD Pro Webcam C910, nameščeno nad poligonom usmerjeno navzdol, kjer z analizo njene slike lahko izračunamo lokacijo in usmeritev robota v poligonu. Kamera je ena prvih HD spletnih kamer za splošen trg, ki zmore ločljivost 1920x1080 slikovnih pik pri 30 sličic na sekundo (FPS). Z večjo ločljivostjo lahko bolj natančno določimo položaj robota v poligonu, z večjo frekvenco sličic pa zmanjšamo zamik med dogodkom v poligonu in prenosom tega dogodka v računalnik in s tem povečamo odzivnost krmilnega programa na dogodke v poligonu.

Ob tej kameri smo vzporedno uporabili tudi spletno kamero Logitech QuickCam Pro 9000 za snemanje samega dogajanja v poligonu.

2.4.2 Linux

Samo ogrodje *Etage* sicer deluje na vseh operacijskih sistemih, ki so podprti s strani GHC prevajalnika za Haskell, kar zaobjema tudi tri trenutno najbolj razširjene operacijske sisteme: Windows, Mac OS X in Linux. V okviru tega dela pa smo za testiranje ogrodja in kot operacijski sistem, ki

služi kot krmilni del robota, uporabili odprtokoden operacijski sistem Linux, ki podpira vse potrebne uporabljene tehnologije, hkrati pa omogoča veliko prilagodljivost glede načina delovanja, tako da se ga da prilagoditi v namenski sistem za krmiljenje robota.

Uporabili smo distribucijo Debian GNU/Linux[8], različico 5.0.6 z jedrom različice 2.6.36[9].

[8] http://www.debian.org/
[9] http://www.kernel.org/

Poglavje 3

Ogrodje za podatkovno pretokovno računanje

Nedeterministično in leno ogrodje za podatkovno pretokovno računanje poimenovano *Etage* je izdelano v obliki knjižnice za programski jezik Haskell. Sestavljeno je iz definicije samega ogrodja kot tudi nekaj osnovnih operacij, ki se lahko uporabljajo samostojno ali kot povezovalne operacije za s strani uporabnika razvite operacije. Tako razvite operacije se lahko povezujejo med seboj in tudi z drugimi ustreznimi operacijami drugih uporabnikov.

Konkretna zasnova delovanja teh uporabnikovih operacij je prepuščena uporabniku: tako lahko operacije omogočajo le zelo osnovno delovanje, na primer aritmetične operacije nad podatki, lahko pa operacije vključujejo kompleksno delovanje sicer imperativnega algoritma oziroma podprograma, ki ga lahko na takšen način transparentno vpnemo med preostale operacije. Osnovne operacije bodo, med ostalim, opisane v tem poglavju, primeri operacij na podlagi ogrodja pa v poglavjema 4 in 5.

Pri razvoju ogrodja smo se odločili za uporabo poimenovanja njegovih elementov po nevroloških oziroma bioloških strukturah, čeprav imajo samo

ogrodje in njegovi elementi le bežno podobnost, tako v delovanju kot strukturi, s temi katerih imena nosijo.

Ogrodje oziroma njegova izvorna koda je javno objavljena[1] pod licenco GPL in s tem na voljo vsem zainteresiranim za uporabo, vpogled v delovanje ali tudi sodelovanje in nadaljnji razvoj.

3.1 Smernice razvoja

Z ogrodjem smo želeli ponuditi sistem za podatkovno pretokovno programiranje, ki bi omogočalo programiranje v obliki izvorne kode. Takšna oblika programiranja namreč omogoča lažjo integracijo z obstoječimi knjižnicami in kodo, hkrati pa se lahko tudi preplete z njo kot v primeru, ko želi uporabnik uporabiti ogrodje le v delu svojega širšega programa. Ob tem omogoča tudi uporabo že obstoječih orodij za razvoj in programiranje.

Kot cilj smo si zastavili, da je ogrodje na voljo uporabnikom v obliki standardne Haskell knjižnice, na podlagi katere lahko uporabniki razvijejo svoje knjižnice, za bolj specifična področja uporabe. Prav tako smo se odločili, da knjižnica ne bo spreminjala oziroma vpeljevala nove sintakse v sam programski jezik, kar sicer Haskell podpira, zaradi česar vsebujejo nekateri konstrukti za uporabo ogrodja dodatne elemente za zadovoljitev obstoječih pravil programskega jezika, je pa zato sama sintaksa brez zakritega delovanja in s tem presenečenj.

Pomemben cilj je bila tudi modularnost in razširljivost ogrodja, za kar smo uporabili široke zmožnosti sistema tipov v Haskellu. Želeli smo si namreč razširljivosti v dve smeri: možnost vpeljave novih funkcionalnosti (funkcij) nad obstoječimi podatkovnimi tipi (oziroma njihovimi družinami) in obra-

[1] `http://hackage.haskell.org/package/Etage`

tno, možnost vpeljave novih tipov podatkov, nad katerimi bi naj delovale obstoječe funkcije. Seveda brez redefinicije obstoječih podatkovnih tipov oziroma funkcij. Če to povzamemo v okviru podatkovno pretokovnega računanja, smo si želeli možnosti zamenjave operacij mreže z drugimi, ne da bi morali v ta namen spremeniti definicijo podatkovnih tipov. In prav tako možnost zamenjave tipa podatkov v mreži, ne da bi morali redefinirati operacije mreže, če že opravljajo željeno funkcionalnost. Še dodatne razsežnosti modularnosti pa smo dosegli z leno zasnovo ogrodja, ki omogoča enostavnejšo ponovno uporabo podprogramov, četudi se uporabljajo le deli le-teh. Podobno kot je opisano v [16] za lene funkcijske programske jezike v splošnem.

Ogrodje je nedeterministično v smislu, da časovna karakteristika pretoka podatkov po mreži ni deterministična sama po sebi oziroma ogrodje ne nudi nobenih zagotovil glede tega, ampak deluje po principu *najboljšega truda* (angl. best effort). Isti podatki lahko naprimer po isti povezavi pri večkratnem zagonu istega programa potujejo različno dolgo in tudi operacije se lahko izvajajo različno dolgo. Zaradi mrežne oblike oblike programa to pomeni, da se lahko različni deli programa izvajajo v različnem časovnem zaporedju pri večkratnem zagonu tega programa. S tem se dosežejo boljše zmogljivosti v primerih, ko determinističnost ni potrebna, saj ni odvečne režijske obremenitve, ki je potrebna za zagotavljanje determinističnosti, prav tako pa se lahko sam program med delovanjem bolje prilagaja trenutnim razpoložljivim virom, ki so seveda lahko različni med zagoni programa. Z drugimi besedami, manj je odvisnosti med različnimi deli programa in s tem je več možnosti za pararelizacijo programa.

Je pa zato potrebna v primeru želje po determinističnem delovanju uporaba ustreznih pristopov oziroma konstruktov pri programiranju. Včasih

možnost nedeterminističnosti v danem programu sploh ne nastopi, sicer pa jo lahko v splošnem (za konkretne primere obstajajo tudi bolj specifične rešitve) dosežemo z uporabo posebnega podatka (žeton), ki s svojim (predvidljivim) prehajanjem po mreži zagotavlja predvidljivo (deterministično) izvajanje mreže. Včasih lahko najdemo vlogo žetona tudi v podatkih že sicer uporabljenih v programu, ne da bi ga nam bilo potrebno umetno vpeljevati.

Ogrodje smo zastavili tudi tako, da pomaga sistemu tipov zagotavljati pravilno delovanje programa s tem, da tipi nosijo dodatno semantično informacijo o programu, oziroma njegovi naravi podatkovno pretokovnega računanja, s čimer pripomorejo k pravilnosti programa že v fazi njegovega prevajanja. Prav tako ogrodje pomaga tudi s pravilno in zagotovljeno inicializacijo in deinicializacijo vmesnikov z zunanjimi napravami ter drugimi imperativnimi deli programa. To omogoča uporabo programov na osnovi tega ogrodja za stabilno interakcijo z realnimi uporabami v fizičnem svetu, naprimer v robotiki.

Sama implementacija tudi abstrahira primitive programskega jezika Haskell tako, da jo je po potrebni kasneje možno zamenjati, ne da bi bilo potrebno zaradi tega spremeniti že razvite programe. Ta abstrakcija se vrši večinoma na nivoju tipov, zato zaradi nje ni dodatne zmogljivostne obremenitve med izvajanjem programa. Trenutno se v implementaciji pri prenašanju podatkov med operacijami uporablja zaklepanje kritičnih sekcij. Lahko pa bi se uporabile tudi sestavljive pomnilniške transakcije [8, 12]. Slednje bi se lahko za potrebe podatkovno pretokovnega računanja izkazalo za primernejše zaradi svoje optimistične narave, saj je hkratnih dostopov do iste kritične sekcije v primeru podatkovno pretokovnega računanja v povprečju malo in zato lahko (pogosto nepotrebno) vsakokratno zaklepanje predstavlja večji zmogljivostni strošek v primerjavi z optimistično izvedbo kritične sek-

cije in v (redkem) primeru konflikta njeno razveljavitev in ponovno izvedbo.

Ob sami osnovni podpori za podatkovno pretokovno računanje ogrodje podpira tudi dogodkovne mreže, oziroma z drugimi besedami dogodkovno pretokovno računanje.

Seveda sta nam bila najpomembnejši vodili učinkovitost in stabilnost delovanja ogrodja, saj si želimo, da deluje tudi v vseh mejnih primerih uporabe. Zaradi svoje narave, ki podpira enostavno in avtomatično vzporedno delovanje, to pomeni delovanje brez nezaželjenih smrtnih objemov in napak pri delu s pomnilnikom in po drugi strani dober izkoristek razpoložljivih (vzporednih) virov.

3.2 Arhitektura ogrodja

Operacije, ki smo jih v okviru ogrodja poimenovali *Neuron*, se povezujejo med seboj s povezavami (v okviru ogrodja, *Nerve*), po katerih se pretakajo podatki (*Impulse*).

3.2.1 Operacije

Na sliki 3.1 je prikazana definicija razreda tipov *Neuron*.

Najprej je naštetih nekaj pogojev za katere podatkovne tipe lahko definiramo instanco tega razreda tipov. V našem primeru za podatkovne tipe, ki so že instance razreda tipov *Typeable*. Ta definira metode, ki omogočajo vpogled v tip vrednosti med delovanjem programa. Zatem povemo, da sta s tem razredom tipov povezana dva tipa, namreč tipa podatkov, ki izstopajo in vstopajo v to operacijo. Glede na predpogoje morata biti instanci razreda tipov *Typeable* in *Impulse*. Zatem sledi še definicija podatkovnega tipa možnih nastavitev operacije, kar omogoča prilagajanje uporabe bolj splošne

```
1  class (Typeable n, Impulse (NeuronFromImpulse n), Impulse (NeuronForImpulse n), Typeable (NeuronFromImpulse n),
2         Typeable (NeuronForImpulse n)) ⇒ Neuron n where
3    type NeuronFromImpulse n
4    type NeuronForImpulse n
5    data NeuronOptions n
6    mkDefaultOptions :: IO (NeuronOptions n)
7    getNeuronMapCapability :: NeuronOptions n → NeuronMapCapability
8    grow :: NeuronOptions n → IO n
9    live :: Nerve (NeuronFromImpulse n) fromConductivity (NeuronForImpulse n) forConductivity → n → IO ()
10   dissolve :: n → IO ()
11   attach :: (NeuronOptions n → NeuronOptions n) →
12             Nerve (NeuronFromImpulse n) fromConductivity (NeuronForImpulse n) forConductivity → IO LiveNeuron
13   mkDefaultOptions = return ⊥
14   getNeuronMapCapability _ = NeuronFreelyMapOnCapability
15   grow _ = return ⊥
16   dissolve _ = return ()
17   live _ _ = waitForException
18   attach = attach′
```

Slika 3.1: Definicija razreda tipov *Neuron*.

operacije.

Sledijo še metode razreda tipov, ki jim moramo, tako kot prej omenjene tipe, ob definiciji nove instance razreda tipov *Neuron* definirati, če nismo zadovoljni s privzetimi definicijami. Metode so podane v dveh delih, najprej v obliki *podpisa tipa* (angl. type signature) metod v vrsticah 6–12, zatem pa sledijo njihove smiselne privzete definicije. Najpomembnejše so metode *grow*, *live* in *dissolve*.

V Haskellu se podpis tipa vrednosti (in funkcij, saj so funkcije v Haskellu tudi vrednosti) poda za imenom vrednosti in znakom ::. Funkcije in v našem primeru metode razreda tipov ponavadi prejemajo več argumentov, ki se jih poda v obliki zaporedja tipov ločenih z znakom $\rightarrow$. Na takšno zaporedje se lahko gleda kot na funkcijo, ki prejema prvi argument in vrača funkcijo preostalih argumentov, kjer se vse skupaj ponovi. V praksi pa si ponavadi predstavljamo, da funkcija prejema kot argumente vse do predzadnjega elementa zaporedja podpisa tipa, zadnji element pa predstavlja tip rezultata funkcije.

Pri sami definiciji funkcije oziroma metode se z _ označijo argumenti, ki jih ne bomo uporabili, kakor smo to naredili v privzetih definicijah metod, ki delujejo splošno, ne glede na argumente. Z $\perp$ se v Haskellu poda vrednost, ki predstavlja dno. Ker je Haskell len programski jezik, takšno vrednost lahko uporabimo v kodi programa, le program je med izvajanjem ne sme uporabiti. V tem primeru program divergira: v teoriji se nikoli ne zaključi, v praksi pa se sproži izjema.

V metodi *grow* se pričakuje, da uporabnik, ki razvija svojo operacijo, ustrezno inicializira *Neuron*. To pomeni, da zasede ustrezne vire monade *IO*, naprimer odpre datoteke ali druge naprave, pripravi kakšne podatkovne strukture in podobno. Vse to je potem potrebno počistiti v metodi *dissolve*.

Za to, da se metoda res v vseh primerih pokliče, poskrbi naše ogrodje.

Glavno delovanje operacije se definira v metodi *live*. Ponavadi se v neskončni zanki (do prihoda izjeme, naprimer ob zaključku programa) čaka na vhodne podatke, se jih obdela, ter pošlje naprej.

Podatki (definiranih tipov) prihajajo in odhajajo po dvosmerni povezavi, ki jo predstavlja *Nerve*. Le tega metoda *attach* priključi na *Neuron* in ob tem poskrbi še za njegovo pravilno inicializacijo in deinicalizacijo. Večinoma uporabniku te metode ni potrebno definirati, saj je privzeta definicija *attach'* zadosti, ampak lahko pa jo, če si želi.

Ob tem *Neuron* podpira možnost, da uporabnik definira kako se naj interna lahka nit izvajanja operacije razporedi med nitmi operacijskega sistema. To je seveda najboljše prepustiti ogrodju oziroma Haskellu samemu, ampak v določenih primerih uporabe kakšnih zunanjih knjižnic, se mora zagotoviti, da se nekaj lahkih niti izvaja v isti niti operacijskega sistema.

3.2.2 Podatki

Definicija razreda tipov podatkov je predstavljena na sliki 3.2. Glede na predpogoje mora biti tip podatka instanca razreda tipov *Typeable* in *Show*. Slednji razred tipov določi izpis vrednosti v tekstovno obliko. Za definicijo instance je potrebno definirati metodi *impulseTime* in *impulseValue*. Namen prve je pri uporabi ogrodja za dogodkovne mreže, kjer metoda vrača absolutni čas nastanka dogodka. Namen slednje metode pa je vračanje vrednosti podatka v enolični obliki za potrebe obdelave podatka v operacijah, ki znajo delovati na vseh podatkovnih tipih (ki so instance razreda tipov *Impulse*, kar pa je že pogoj za instanco razreda tipov *Neuron*).

Podatki so definirani kot razred tipov zato, da lahko kasneje dodajamo nove tipe vrednosti podatkov in da operacije, ki delujejo nad podatki, in-

```
class (Show i, Typeable i) ⇒ Impulse i where
  impulseTime :: i → ImpulseTime
  impulseValue :: i → ImpulseValue
```

Slika 3.2: Definicija razreda tipov *Impulse*.

```
class Impulse a ⇒ AudioImpulse a where
  samplingResolution :: a → Int
  samplingFrequency :: a → Int
```

Slika 3.3: Definicija razreda tipov *AudioImpulse*.

stancami *Impulse*, brez redefinicije delujejo tudi na njih.

Seveda lahko uporabnik definira tudi svoje razrede tipov podatkov in za tem operacije nad njimi. Naprimer razred podatkov, ki predstavljajo vzorec zvočnega signala, je predstavljen na sliki 3.3. Tako je tip vrednosti, ki je instanca *AudioImpulse*, zaradi predpogoja tudi instanca *Impulse* in ima zanj definirane vsaj metode *samplingResolution*, *samplingFrequency*, *impulseTime* in *impulseValue*.

Pomembno se je zavedati, da je možno ustvariti nove instance razredov tipov tudi kasneje, čisto drugje v programu ali kar v kakšni knjižnici. Torej je možno dodati nov tip vrednosti med podprte s strani operacij nad njimi. In obratno, ob definiciji novega razreda tipov lahko definiramo njegove instance tudi nad tipi vrednosti, ki so definirani že od prej, ne da bi jih morali redefinirati.

Tipe vrednosti, ki so instance *Impulse*, se lahko pri definiciji instance *Neuron*a uporabi za povezana tipa izhodnih, *NeuronFromImpulse n*, in vhodnih podatkov, *NeuronForImpulse n*. Različne operacije, *Neuron*i, lahko tako delujejo nad istimi tipi podatkov.

Lahko pa delujejo tudi splošno nad vsemi tipi, ki so instance razreda

```
data AnyImpulse where
  AnyImpulse :: Impulse i ⇒ i → AnyImpulse
```

Slika 3.4: Definicija eksistencialno kvantificiranega podatkovnega tipa *AnyImpulse*.

tipov *Impulse* (in podobno za *AudioImpulse* ter druge uporabnikove razrede tipov). V ta namen je že vnaprej definiran ustrezni podatkovni tip *AnyImpulse*, ki jih v sistemu tipov združuje s pomočjo *eksistencialno kvantificiranega tipa* (angl. existentially quantified type). Njegova definicija v obliki *generaliziranega algebraičnega podatkovnega tipa* (angl. generalized algebraic data type, GADT) [3, 27] je na voljo na sliki 3.4.

Do sedaj smo torej opisali dva načina definicije tipa podatkov, nad katerimi operacija lahko deluje: eksplicitni tip podatkov in eksistencialno kvantificiran tip podatkov, ki zaobjema vse tipe podatkov, ki so instance razreda tipov *Impulse* (kar v našem primeru predstavlja vse tipe podatkov, nad katerimi lahko delujejo operacije). Slabost slednjega pristopa je, da nad njimi lahko operacije uporabljajo le metode razreda tipov *Impulse* in ne morebitnih drugih metod, ki jih konkretni podatkovni tip tudi podpira.

V programskih jezikih z dinamičnim preverjanjem tipov (ob izvajanju programa) oziroma programskih jezikih, ki dovoljujejo *pretvorbo* (angl. cast) tipov, se ponavadi z uporabo vpogleda v tip vrednosti med delovanjem programa in ustrezno prilagoditvijo tipa doseže dostop do metod za ta bolj specifičen tip. Problem tega pristopa je, da je težko razširljiv ob vpeljavi novih tipov brez spremembe kode na mestu uporabe in da deluje le v kolikor je prilagoditev tipa sploh možna. Predvsem pa da se ob prevajanju pravilnost programa oziroma tipov na mestu prilagoditve izgubi zagotovilo o pravilnosti programa.

```
class (Impulse i, Impulse j) ⇒ ImpulseTranslator i j where
  translate :: i → [j]
```

Slika 3.5: Definicija razreda tipov *ImpulseTranslator*.

Standarden Haskell pretvorbe tipov ne podpira oziroma dovoljuje, kar vidimo kot nekaj pozitivnega, saj želimo, da se že ob prevajanju našega programa preveri, če je so povezave med operacijami v mreži pravilno postavljene in da znajo res vse operacije delovati na podatkih oziroma s podatki, ki jih prejemajo. Torej da so vse poti podatkov dobro definirane, ne pa da se zmota glede tega ugotovi šele med delovanjem programa, ko bi v kompleksni mreži do operacije prišel podatek nepričakovanega tipa.

Zato smo definirali poseben razred tipov, oziroma parov tipov, ki definira metodo *translate* namenjeno preslikavi vrednost enega tipa v seznam vrednosti drugega tipa. Definicija ja na sliki 3.5.

To metodo lahko tako uporabimo povsod, kjer želimo preslikati vrednost podatka enega tipa v vrednost drugega tipa. Haskell že ob prevajanju preveri, če je takšna preslikava definirana. V kolikor ni, jo lahko tudi kasneje enostavno dodamo z definicijo ustrezne instance razreda tipov *ImpulseTranslator* za ustrezni par tipov. In podobno velja tudi za kasneje definirane nove tipe, ki jih mogoče vpeljemo v program oziroma uporabimo kakšno knjižnico, ki jih definira.

V primeru našega ogrodja se ta metoda uporablja ob pošiljanju podatkov po povezavi, kjer Haskell avtomatično izbere njeno ustrezno definicijo glede na vhodni in izhodni tip povezave, oziroma javi napako, če ustrezna definicija ni na voljo. Na takšen način je med seboj mogoče povezovati operacije, ki sicer delujejo na različnih tipih podatkov, hkrati pa ohranimo preverjanje pravilnosti programa s preverjanjem tipov že ob prevajanju in omogočamo

```
instance Impulse i ⇒ ImpulseTranslator i i where
  translate i = [i]
instance Impulse i ⇒ ImpulseTranslator i AnyImpulse where
  translate i = [AnyImpulse i]
```

Slika 3.6: Dve osnovni instanci razreda tipov *ImpulseTranslator*.

bolj kompleksne preslikave med tipi podatkov, kot jih ponavadi omogoča pretvorba tipov.

Iz definicije *ImpulseTranslator* sledita dve osnovni instanci, ki sta predstavljeni na sliki 3.6. Prva predstavlja identiteto, torej vsak tip se lahko preslika v samega vase. Druga pa na podlagi definicije podatkovnega tipa *AnyImpulse* definira preslikavo vanj, ki je po definiciji možna za prav vse tipe, ki so instance razreda tipov *Impulse*. Slednje pomeni, da tudi za operacije, ki delujejo nad tem eksistencialno kvantificiranim tipom podatkov *AnyImpulse*, ogrodje avtomatično preslika vhodne podatke vanj.

Ob tipu *AnyImpulse* je definiranih še nekaj drugih tipov, predstavljenih na sliki 3.7. *NoImpulse* predstavlja tip, ki se ga lahko uporabi za vhodni ali izhodni tip operacije takrat, ko operacija nima vhodnih oziroma izhodnih podatkov. *IValue* predstavlja generični tip podatka, ki nosi eno vrednost (ob času podatka oziroma dogodka). *IList* pa podobno nosi seznam vrednosti. *IInteger*, *IRational*, *IIntegerList* in *IRationalList* pa so pomožna imena tipov za enostavnejšo uporabo, kjer so vrednosti tipa *Integer* oziroma *Rational*.

3.2.3 Povezave

Do sedaj smo definirali podatke in operacije kot dva razreda tipov in s tem omogočili razširljivost in modularnost. Povezave pa so definirane kot

```
data NoImpulse
instance Impulse NoImpulse where
  impulseTime _ = ⊥
  impulseValue _ = ⊥

data (Real r, Show r, Typeable r) ⇒ IValue r = IValue {
  impulseValueTimestamp :: ImpulseTime,
  value :: r
  } deriving (Eq, Ord, Read, Show, Typeable, Data)
instance (Real r, Show r, Typeable r) ⇒ Impulse (IValue r) where
  impulseTime IValue {impulseValueTimestamp} = impulseValueTimestamp
  impulseValue IValue {value}                = [toRational value]
type IInteger  = IValue Integer
type IRational = IValue Rational

data (Real r, Show r, Typeable r) ⇒ IList r = IList {
  impulseListTimestamp :: ImpulseTime,
  list :: [r]
  } deriving (Eq, Ord, Read, Show, Typeable, Data)
instance (Real r, Show r, Typeable r) ⇒ Impulse (IList r) where
  impulseTime IList {impulseListTimestamp} = impulseListTimestamp
  impulseValue IList {list}                = map toRational list
type IIntegerList  = IList Integer
type IRationalList = IList Rational
```

Slika 3.7: Dodatne instance razreda tipov *ImpulseTranslator*.

navaden podatkovni tip *Nerve*, ki sicer abstrahira svoje interno delovanje in omogoča morebitne kasnejše izboljšave oziroma spremembe, ampak ne omogoča pa le-teh s strani uporabnika ogrodja. Razlog za to je predvsem enostavnejša uporaba ogrodja, saj je manj "spremenljivih" delov, še posebej, ker je podatkovni tip *Nerve* že sam po sebi precej kompleksen. Predvsem pa gre morebitne spremembe delovanja povezav, ki bi si jih lahko uporabnik želel, kot so zakasnitve prenosa podatkov, filtriranje podatkov, drugačno razvrščanje čakajočih podatkov v vrsti in podobno, implementirati kot operacijo, ki se povezuje v mrežo normalno preko povezav tipa *Nerve*. To naredi programe lepše in lažje razumljive, saj vsa logika programa ostane v opera-

```
data Nerve from fromConductivity for forConductivity
```

Slika 3.8: Definicija podatkovnega tipa *Nerve*.

cijah, kar je tudi v skladu s pristopom podatkovno pretokovnega računanja.

Vidna definicija podatkovnega tipa *Nerve* je na sliki 3.8. *Nerve* predstavlja dvosmerno povezavo, kjer lahko za vsako smer (le-to smo poimenovali *Axon*) definiramo tip podatkov, ki se po njej lahko prenašajo. To predstavljata tipa *from* in *for*, ki morata biti instanci razreda *Impulse*. *from* predstavlja tip podatkov iz smeri operacije, *for* pa v smeri operacije. *fromConductivity* in *forConductivity* določata za vsako smer, če bo ta smer v programu v uporabi, za kar uporabimo prazna podatkovna tipa *AxonConductive* in *AxonNonConductive*. Ta dodatna informacija v tipu *Nerve* pomaga pri preprečevanju morebitnega nezaželjenega polnjenja pomnilnika, kjer bi kakšna operacija pošiljala podatke po povezavi, na drugi strani pa druga operacija teh podatkov ne bi brala. Prav tako to omogoči, da se ob prevajanju preverijo tudi morebitne napake pri povezovanju mreže, naprimer branje iz povezave, ki smo jo označili, da je ne bomo uporabljali z *AxonNonConductive*.

Vsaka smer, *Axon*, deluje kot FIFO vrsta. Tako je pomembno, da ga operacija disciplinirano bere, saj lahko sicer pride do polnjenja vrste in s tem zakasnjevanja podatkov. Odvisno od namena programa lahko to predstavlja problem. Ker pa so vsi podatki opremljeni tudi s časom svojega nastanka (za njihovo uporabo kot dogodke), lahko operacija to opazi in se ustrezno odzove. V praksi se sicer izkaže, da ali za program ni problem, če se celotno njegovo izvajanje zakasni (nima zahtev po realno-časoven delovanju) in zato lahko operacija bere podatke po vrsti, ali pa se program mora odzivati hitro,

```
sendForNeuron :: Nerve from fromConductivity for AxonConductive → for → IO ()
getFromNeuron :: Nerve from AxonConductive for forConductivity → IO from
maybeGetFromNeuron :: Nerve from AxonConductive for forConductivity → IO (Maybe from)
slurpFromNeuron :: Nerve from AxonConductive for forConductivity → IO [from]
waitAndSlurpFromNeuron :: Nerve from AxonConductive for forConductivity → IO [from]
getContentsFromNeuron :: Nerve from AxonConductive for forConductivity → IO [from]
sendListForNeuron :: Nerve from fromConductivity for AxonConductive → [for] → IO ()
```

Slika 3.9: Definicije funkcij za delo s povezavo zunaj operacije.

```
sendFromNeuron :: Nerve from fromConductivity for forConductivity → from → IO ()
getForNeuron :: Nerve from fromConductivity for forConductivity → IO for
maybeGetForNeuron :: Nerve from fromConductivity for forConductivity → IO (Maybe for)
slurpForNeuron :: Nerve from fromConductivity for forConductivity → IO [for]
waitAndSlurpForNeuron :: Nerve from fromConductivity for forConductivity → IO [for]
getNewestForNeuron :: Data for ⇒ Nerve from fromConductivity for forConductivity → IO [for]
getContentsForNeuron :: Nerve from fromConductivity for forConductivity → IO [for]
sendListFromNeuron :: Nerve from fromConductivity for forConductivity → [from] → IO ()
```

Slika 3.10: Definicije funkcij za delo s povezavo znotraj (v definiciji) operacije.

v realnem času, pri čemer pa ni problem, če se kakšen podatek izpusti (preskoči). Tako smo za branje iz povezave definirali dva tipa funkcij: funkcije, ki berejo vse podatke po vrsti in funkcije, ki izpustijo (preskočijo) vse stare podatke in preberejo le najnovejšega. Seveda pa lahko uporabnik po potrebi definira tudi lastne, bolj kompleksne oblike branja podatkov iz povezave. Definirane funkcije so naštete na slikama 3.9 in 3.10.

Večinoma podatkovnega tipa povezave niti ni potrebno eksplicitno uporabljati, saj tudi sam konstruktur podatkovnega tipa ni direktno na voljo, ampak se povezave zgradijo skupaj s celotno mrežo v posebnem okolju ogrodja poimenovanem *Incubation*. Sicer je za potrebe morebitnih posebnih mrež na voljo funkcija *growNerve*, katere uporaba pa se za splošne mreže odsvetuje.

Več o tem in okolju *Incubation* v naslednjem, 3.2.4, razdelku.

3.2.4 Gradnja mreže

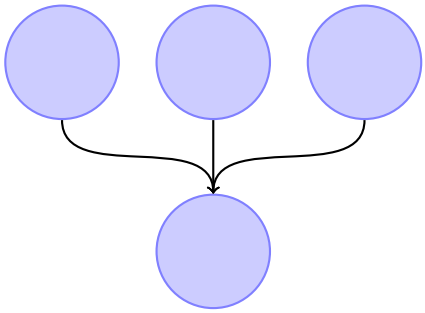
Ker je gradnja mreže, torej povezovanje operacij med seboj, najpomembnejši del podatkovno pretokovnega programa, seveda ob sami definiciji operacij, ima ogrodje *Etage* posebno okolje za definicijo oziroma gradnjo mreže programa. To okolje, ob samem preverjanju pravilnosti delovanja, ki se lahko ugotovi ob prevajanju, omogoča tudi pravilno gradnjo, delovanje in tudi razgradnjo mreže med samim delovanjem programa. Poimenovali smo ga *Incubation*.

Okolje *Incubation* tako v osnovi skrbi za ustvarjanje operacij in njihovo povezovanje med seboj na dva osnovna načina. V ozadju pa poskrbi, da se vse operacije res uspešno ustvarijo ter ob zaključku programa tudi počistijo, tudi v primeru napake oziroma izjeme. Hkrati še dodatno preveri (kot nadgradnja statične analize ob prevajanju), če se vsi podatki, ki se pošiljajo po povezavah, tudi res berejo, da ne bi med delovanjem programa prišlo do nezaželjenega polnjenja pomnilnika. Slednje bi se sicer s še bolj kompleksno uporabo (aritmetike) tipov naredilo že ob prevajanju, a trenutno uradna podpora za to še ni stabilizirana v prevajalniku[2]. Okolje tudi poskrbi za pravilno večkratno razpošiljanje podatkov po povezavi, če je na njo priključenih več operacij.

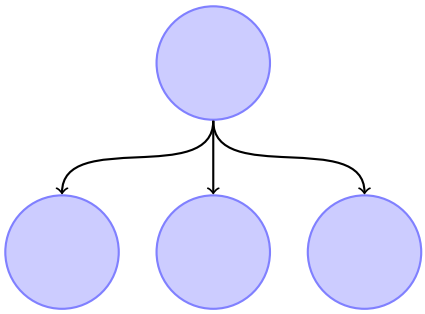
Slika 3.11 prikazuje operacijo, ki na svojo vhodno povezavo prejema podatke iz treh drugih operacij. Ker povezave delujejo kot FIFO vrsta, se preprosto vse podatki pišejo v isto povezavo.

Iz slike 3.12 je razvidno, da morajo biti podatki, ki zapuščajo operacijo po izhodni povezavi, na voljo trem operacijam. Če bi v tem primeru upora-

[2] `http://hackage.haskell.org/trac/ghc/wiki/TypeNats`



Slika 3.11: Primer mreže, kje na isto povezavo pošilja podatke več operacij.



Slika 3.12: Primer mreže, kje se iz iste povezave pošilja podatke na več operacij.

bljali povezavo le kot navadno FIFO vrsto, bi podatek lahko prebrala le prva operacija, ki bi se branja lotila. Zato je potrebno ustrezno poskrbeti, da se povezave pravilno cepijo, torej da je podatek na voljo vsem operacijam, ki so povezane na povezavo, in to po za njih FIFO principu.

Okolje *Incubation* je implementirano kot monada, zaradi česar ga lahko uporabnik uporablja v do-zapisu. In ker je hkrati to začetni stik uporabnika z okoljem, to pomeni, da mu je samo delo z ogrodjem lahko bolj domače, kot bi mu bilo siceršnjo čisto funkcijsko. Vse monade oziroma instance razreda tipov *Monad* morajo zadoščati trem zakonom: levi identiteti, desni identiteti in asociativnosti. Načrtovati lastno monado, ki bi delovala kot si želimo in hkrati zadoščala tem pogojem lahko hitro postane težavno, še posebej

pri kompleksnejših monadah, katerih logika ne temelji na kakšni poznani matematični analogiji. V našem primeru si želimo monado, ki bo znala zaporedoma graditi mrežo povezav in operacij in bo v primeru, da pride pri tem do napake oziroma izjeme, znala ustrezno počistiti do takrat zgrajeno mrežo. Kako točno si v tem primeru predstavljati omenjene zakone mogoče ni najbolj jasno.

Na srečo se je sčasoma izkazalo, da če si monade predstavljamo skozi operacijsko semantiko, jih lahko načrtujemo in implementiramo precej sistematično, za kar je na voljo tudi knjižnica [1]. Zanjo je tako potrebno definirati željene operacije, v našem primeru: *NeuronOperation*, *AttachOperation* in *FuseOperation*, ki so potem v okolju *Incubation* na voljo skozi funkcije *growNeuron*, *attachTo* in *fuseWith*, katerih podpisi tipa so na sliki 3.15.

Funkcija *growNeuron* sprejme kot argument funkcijo, ki nastavi možne nastavitve glede na prejete privzete nastavitve, in vrne povezavo, *Nerve*, ki jo priključi na inicializirano operacijo, *Neuron*. Glede na tip dobljene povezave lahko določimo tipa *fromConductivity* in *forConductivity*.

Funkcija *attachTo* sprejme povezavo in njen izhod pravilno priključi na vhode naštetih v seznamu povezav. To naredi tako, da se vsi podatki pošiljajo res na vse naštete povezave in ne le naprimer na prvo, ki podatek prebere. V seznamu uporabljamo eksistencialno kvantificiran tip *TranslatableFor*, ki predstavlja vse povezave, katerih tipi podatkov se lahko preslikajo (z uporabo razreda tipov *ImpulseTranslator*) iz danega tipa podatkov, torej tipa podatkov povezave iz katerih se podatki posredujejo na naštete povezave. S tem dosežemo, da Haskell med prevajanjem zagotovi, da lahko povezujemo le povezave, za katere obstaja ustrezna instanca razreda tipov *ImpulseTranslator*. Ta funkcija je osnovna funkcija s katero povezujemo povezave med seboj in jo zato lahko zasledimo v skoraj vseh programih na

```
1 main = do
2   prepareEnvironment
3   incubate $ do
4     nerveRandom ← (growNeuron :: NerveOnlyFrom (SequenceNeuron Int)) defaultOptions
5     nerveDump ← (growNeuron :: NerveOnlyFor DumpNeuron) defaultOptions
6     nerveRandom `attachTo` [TranslatableFor nerveDump]
```

Slika 3.13: Primer uporabe monade *Incubation*.

```
1 incubate $ do
2   nerveRandom1 ← (growNeuron :: NerveOnlyFrom (SequenceNeuron Int)) defaultOptions
3   nerveRandom2 ← (growNeuron :: NerveOnlyFrom (SequenceNeuron Int)) defaultOptions
4   nerveDump ← (growNeuron :: NerveOnlyFor DumpNeuron) defaultOptions
5   nerveFused ← [TranslatableFrom nerveRandom1, TranslatableFrom nerveRandom2]
6     `fuseWith` (impulseFuser ((:[]) ∘ sum ∘ concat))
7   nerveFused `attachTo` [TranslatableFor nerveDump]
```

Slika 3.14: Primer uporabe funkcije *fuseWith*,
ki sešteva pare vrednosti iz para vhodnih povezav.

osnovi našega ogrodja, recimo na primeru na sliki 3.13.

Funkcija *fuseWith* pa iz danega seznama povezav bere po en podatek iz vsake povezave, nad tako zbranimi podatki pokliče dano funkcijo in rezultat pošlje po povezavi, ki jo *fuseWith* vrne. Zatem vse skupaj ponovi. Torej osnovna ideja te funkcije je, da se lahko iz več povezav skupaj obdelujejo podatki. Pomembna lastnost je njeno sinhronizacijsko delovanje, saj zahteva natanko po en podatek iz vsake podane povezave in do takrat nanj čaka. Tako se zagotovi, da obdeluje vedno podatke iz vseh povezav. Je pa zato potrebno paziti, da imajo vse povezave približno enako gostoto podatkov, saj bi se sicer v nekaterih povezavah kopičili podatki. Pri tej funkciji v seznamu uporabljamo eksistencialno kvantificiran tip *TranslatableFrom*, ki

predstavlja vse povezave, katerih podatki se lahko preslikajo v isti, podan, tip podatkov. Ta način vezave povezav se ne uporablja tako pogosto, saj lahko večkrat dosežemo enak učinek s pravilno vezavo povezav s funkcijo *attachTo*, po drugi strani pa uporaba funkcije *fuseWith* izboljša preglednost in razumevanje programa. Primer uporabe je na sliki 3.14.

Iz monade *IO* se lahko povzdignemo v našo monado *Incubation*, torej v naše okolje za gradnjo mreže, z uporabo funkcije *incubate*. Ob njej je za lažje definiranje tipa povezave, ki jo vrne funkcija *growNeuron*, definiranih tudi nekaj pomožnih tipov (oziroma bolj natančno funkcij tipov, ki vračajo ustrezen tip). Le ti so našteti na sliki 3.16.

Verjetno je najlažje predstaviti njihovo uporabo in funkcijo *incubate* kar na primeru na sliki 3.13. Funkcija *main* je vstopna funkcija v programskem jeziku Haskell in zavzema vrednosti v monadi *IO*. Najprej poženemo funkcijo *prepareEnvironment*, ki nastavi pravilno obdelavo signalov operacijskega sistema, da se program pravilno zaključi. Zatem v okolju *Incubation* ustvarimo dve operaciji, *SequenceNeuron*, ki vrača seznam naključnih vrednosti tipa *Int*, in *DumpNeuron*, ki katerikoli dobljen podatek izpiše uporabniku na terminal. Za obe operaciji uporabimo kar privzete nastavitve. In na koncu povežemo izhod povezave prvega na vhod povezave drugega, tako da se naključne vrednosti, ki se generirajo, izpisujejo uporabniku.

V tem kratkem programu lahko vidimo uporabo eksistencialno kvantificiranega tipa *TranslatableFor*. Uporabimo tudi tipa *NerveOnlyFrom* in *NerveOnlyFor*, kjer prvi pove, da smo zainteresirani le za poslane podatke s strani operacije, drugi pa le za posredovanje podatkov operaciji. S tem se ustrezno nastavijo tipi povezave, ki določajo kako delujejo povezave, pripete na operacije, ter s tem omogočijo preverjanje pravilnosti mreže ob prevajanju in ob izgradnji mreže ob zagonu programa.

```
1 growNeuron :: (Neuron n, GrowAxon (Axon (NeuronFromImpulse n) fromConductivity),
2                                   GrowAxon (Axon (NeuronForImpulse n) forConductivity)) ⇒
3   (NeuronOptions n → NeuronOptions n) → Incubation (Nerve (NeuronFromImpulse n) fromConductivity
4                                                           (NeuronForImpulse n) forConductivity)
5 attachTo :: ∀from for forConductivity.(Impulse from, Impulse for) ⇒
6   Nerve from AxonConductive for forConductivity → [TranslatableFor from] → Incubation ()
7 fuseWith :: ∀i j.(Impulse i, Impulse j) ⇒
8   [TranslatableFrom i] → (ImpulseTime → [i] → [j]) → Incubation (Nerve (FuseFromImpulse i j) AxonConductive
9                                                                        (FuseForImpulse i j) AxonNonConductive)
```

Slika 3.15: Definicije funkcij, ki ustrezajo operacijam okolja oziroma monade *Incubation*.

```
type NerveBoth n = (NeuronOptions n → NeuronOptions n) →
  Incubation (Nerve (NeuronFromImpulse n) AxonConductive (NeuronForImpulse n) AxonConductive)
type NerveNone n = (NeuronOptions n → NeuronOptions n) →
  Incubation (Nerve (NeuronFromImpulse n) AxonNonConductive (NeuronForImpulse n) AxonNonConductive)
type NerveOnlyFrom n = (NeuronOptions n → NeuronOptions n) →
  Incubation (Nerve (NeuronFromImpulse n) AxonConductive (NeuronForImpulse n) AxonNonConductive)
type NerveOnlyFor n = (NeuronOptions n → NeuronOptions n) →
  Incubation (Nerve (NeuronFromImpulse n) AxonNonConductive (NeuronForImpulse n) AxonConductive)
```

Slika 3.16: Pomožni tipi za uporabo ob funkciji *growNeuron*.

Kateri *Neuron* naj funkcija *growNeuron* sploh inicializira in poveže na povezave se prav tako določi s specifikacijo tipa. Haskellov prevajalnik pa ob prevajanju izbere pravilno definicijo oziroma instanco razreda tipov *Neuron*.

Ogrodje daje za potrebe morebitnih posebnih mrež na voljo tudi osnovne funkcije okoli katerih je to okolje sestavljeno. In sicer funkcije *growNerve*, *propagate*, *fuse*, *branchNerveFor*, *branchNerveFrom*, *branchNerveBoth* ter *cross*. Uporaba teh se za splošne mreže odsvetuje, saj je potem prepuščeno uporabniku, da poskrbi za njihovo pravilno uporabo.

3.3 Osnovne operacije

V razdelku 3.2.4 smo v primeru na sliki 3.13 že uporabili nekaj osnovnih operacij, ki so na voljo z ogrodjem. Te služijo kot primeri uporabnikom, za lažjo definicijo lastnih operacij, kot tudi za nekaj pogostih operacij, ki pridejo prav ob podatkovno pretokovnem programiranju.

3.3.1 Generator zaporedja

Operacija *Sequence* omogoča definirati generator zaporedja podatkov željenega tipa kot tudi zakasnitve med njimi. Največkrat se uporablja za generiranje naključnih podatkov z naključnimi zakasnitvami med njimi, oboje v nekem intervalu možnih vrednosti. Ta operacija je primer operacije, kjer je izhodni tip parametriziran. Nastaviti je možno:

valueSource :: $[v]$

: Seznam (ponavadi neskončen) vrednosti podatkov, ki jih naj operacija pošlje. Privzet je naključen neskončen seznam vrednosti tipa v.

intervalSource :: $[Int]$

$$nerveRandom \leftarrow (growNeuron :: NerveOnlyFrom\ (SequenceNeuron\ Int))\ defaultOptions$$

$$nerveOnes \leftarrow (growNeuron :: NerveOnlyFrom\ (SequenceNeuron\ Int))\ (\lambda o.\ o\ \{valueSource = repeat\ 1\})$$

Slika 3.17: Primera uporabe operacije *Sequence*.

Seznam zakasnitev med vrednostmi pri pošiljanju po povezavi. Privzet je seznam največ 1 sekundo dolgih naključnih zakasnitev.

Na sliki 3.17 sta prikazana dva primera uporabe. Prvi predstavlja privzeto delovanje, torej generiranje naključnih vrednosti tipa *Int* z največ 1 sekundo dolgimi naključnimi zakasnitvami med njimi. Drugi primer pa za seznam vrednosti *valueSource* nastavi neskončno zaporedje enic.

3.3.2 Izpisovanje

Operacija *Dump* izpisuje vse podatke, ki jih dobi po svoji povezavi, uporabniku na terminal. Primer je na sliki 3.18. Ta operacija je primer operacije, ki lahko sprejema poljuben tip podatkov. Na voljo so sledeče nastavitve:

handle :: *Handle*

Kam se naj podatki izpisujejo. Privzet je standardni izhod.

showInsteadOfDump :: *Bool*

Namesto izpisovanja vrednosti *impulseTime* in *impulseValue*, se naj uporabi standardna funkcija *show*.

prefix :: *String*

Predpona pri izpisovanju. Po privzetem je ni.

$$nerveDump \leftarrow (growNeuron :: NerveOnlyFor\ DumpNeuron)\ (\lambda o.\ o\ \{showInsteadOfDump = True\})$$

Slika 3.18: Primer uporabe operacije *Dump*.

$$nerveFunction \leftarrow (growNeuron :: NerveBoth\ (FunctionNeuron\ AnyImpulse\ IRational))\ (\lambda o.\ o\ \{function = \lambda t.\ (:[]) \circ IValue\ t \circ sum \circ impulseValue\})$$

Slika 3.19: Enostaven primer operacije *Function*.
Seštevanje *impulseValue* vrednosti podatka.

3.3.3 Apliciranje funkcije

Za apliciranje funkcije na vse podatke, ki pridejo po povezavi, je na voljo operacija *Function*. Ker je Haskell len programski jezik, to ne pomeni, da bo rezultat takoj (do konca) evaluiran, ampak šele ko (in če) bo rezultat potreben (verjetno v kakšni drugi operaciji). Je primer operacije, kjer sta tako vhodni kot izhodni tip parametrizirana. Nastavitev je le ena:

$function :: ImpulseTime \rightarrow i \rightarrow [j]$

> Funkcija, ki se naj aplicira nad vhodnimi podatki. Vrne seznam izhodnih podatkov, ki se pošljejo v vrstnem redu seznama. Privzeta funkcija vedno vrača prazen seznam.

Ta operacija je uporabna takrat, ko želimo na podatke aplicirati čisto funkcijo in z uporabo te operacije se lahko izognemo večini primerov definicije lastne operacije v ta namen. Enostaven primer uporabe je na sliki 3.19, kompleksnejši pa na sliki 3.20. V primeroma smo uporabili kar osnovne vnaprej pripravljene tipe podatkov, *IRational* in *IIntegerList*, seveda pa lahko uporabnik uporabi v operaciji poljubne svoje vhodne in izhodne tipe podatkov.

```
1  incubate $ do
2    let gcd t IList {list = (a : b : is)} = let r = a `mod` b in
3                                            if r ≡ 0
4                                              then [IList t (b : is)]
5                                              else [IList t (b : r : is)]
6        gcd _ _                           = []
7    nerveDump ← (growNeuron :: NerveOnlyFor DumpNeuron)
8                  (λo. o {showInsteadOfDump = True})
9    nerveSum ← (growNeuron :: NerveBoth (FunctionNeuron IIntegerList IIntegerList))
10                 (λo. o {function = gcd})
11   nerveSum `attachTo` [TranslatableFor nerveSum, TranslatableFor nerveDump]
12   liftIO $ do
13     t ← getCurrentImpulseTime
14     sendForNeuron nerveSum $ IList t [110, 80, 5]
```

Slika 3.20: Kompleksnejši primer operacije *Function*.
Računanje največjega skupnega delitelja.

Operacija na sliki 3.19 sešteva vse vrednosti, ki jih na vhodnih podatkih vrne metoda *impulseValue*. Na sliki 3.20 pa imamo program, ki za vhodni podatek, ki predstavlja seznam števil, iterativno izračuna največji skupni delitelj teh števil. Tako se v vsaki iteraciji izvede naslednji korak Evklidovega algoritma, ki se po povratni zanki vrne nazaj, hkrati pa tudi izpiše. Zadnja pa kot primer zažene izračun največjega skupnega delitelja števil 110, 80 in 5.

3.3.4 Zakasnitev

Enostaven način shranjevanja podatkov oziroma stanja v mreži se lahko doseže z uporabo operacije *Delay*, ki le zakasni vse podatke, ki jih prejme, za nastavljeno število novih korakov. Podatkovno pretokovni programi v splošnem nimajo ure, ki bi krmila proženje operacij, ampak je to odvisno od prihoda podatkov. Zato je primerljiva zakasnitvi (oziroma hranjenju)

podatkov za določeno število urnih ciklov v ukazno pretokovnih programih naša zakasnitev za določeno število vhodnih podatkov.

Je primer operacije, ki na posreden način vpliva na to, kako se podatki prenašajo po mreži. Sicer še vedno deluje po FIFO principu, ampak kakšne sorodne operacije bi lahko delovale tudi drugače. Zaradi te podobnosti s FIFO principom to tudi pomeni, da se pogosto lahko s pravilno sestavljeno in inicializirano mrežo izognemo uporabi te operacije, saj lahko izkoristimo podobno naravo shranjevanja podatkov v čakajočo vrsto kar povezav samih.

delay :: *Int*

Za koliko vhodnih podatkov naj bodo izhodni podatki zakasnjeni.

3.3.5 Delavec

Včasih si želimo asinhrone interakcije mreže z okoljem. Naprimer, da si želimo notranje delovanje mreže na nek način vizualizirati, hkrati pa si ne želimo, da ta vizualizacija vpliva na samo delovanje mreže. Tu je lahko koristna operacija *Worker*, katere namen je, da sprejema podatke, ki predstavljajo akcije monade *IO*, ter jih izvede. Zaradi vzporedne narave podatkovno pretokovnega računanja, to pomeni, da se tudi ta akcija izvede neodvisno od preostalega delovanja mreže, ki lahko neodvisno še naprej deluje.

Ta operacija kaže, da so podatki, ki se pošiljajo po povezavah, lahko precej drugačni od tega, kar si ponavadi predstavljamo. Torej v tem primeru kar akcije monade *IO* same. Seveda bi lahko pošiljali tudi kar mreže same. Da je to možno, se imamo zahvaliti naravi Haskella, kjer so funkcije prvorazredne vrednosti.

mapOnCapability :: *NeuronMapCapability*

Včasih je potrebno več akcij monade *IO* izvajati v isti niti operacijskega

sistema, kar se lahko nastavi z uporabo te nastavitve.

Trenutno operacija le sprejema podatke in izvaja akcije, ki morajo biti zaradi tega tipa *IO* (). Lahko pa bi se operacija razširila tudi tako, da bi lahko akcija vrnila vrednost, ki bi se zatem poslala naprej kot izhodni podatek operacije.

Poglavje 4

Primer uporabe: računanje najcenejše poti v grafu

Do sedaj smo predstavili ogrodje samo in nekaj osnovnih, že definiranih, operacij, ki lahko služijo kot primer že sami po sebi. Seveda pa se prava vrednost ogrodja pokaže šele takrat, ko ga uporabimo za reševanje konkretnih problemov. V ta namen ga bomo sedaj predstavili na primeru računanja najcenejše poti v grafu, ki bo hkrati tudi primer delovanja ogrodja na velikem številu operacij, v poglavju 5 pa nadaljevali z realnim problemom krmiljenja robota, kjer bo tudi število uporabljenih operacij manjše. Koda programa uporabljenega v tem primeru se nahaja v dodatku A.

Večinoma so trenutno znani in uporabljeni algoritmi za računanje najcenejše poti v grafu po naravi imperativni. Uporaba funkcijskih algoritmov je redka [11]. A po drugi strani so že sami podatkovno pretokovni programi v osnovi podani v obliki grafov, torej z operacijami kot vozlišči in povezavami med njimi. Zato se postavi vprašanje, ali bi lahko naravo teh programov izkoristili tudi za računanje samih lastnosti grafa. V tem primeru bi lahko graf nad katerim želimo računati njegove lastnosti preslikali izomorfno v program,

same operacije oziroma vozlišča pa bi izvajala željen izračun.

Za primer smo si izbrali računanje najcenejše poti v grafu, saj je eden osnovnih algoritmov, ki jih ponavadi delamo nad grafi. Glede tega, kako naj bi potekalo samo izvajanje operacij, se lahko zgledujemo po imperativnih algortmih po eni strani ter implementaciji le-teh v distribuiranih sistemih, naprimer v računalnikših omrežjih. V naši implementaciji smo se tako zgledovali po algoritmu, kot je v uporabi v usmerjevalnem protokolu v računalniških omrežjih Babel [4].

Primer smo razvili v obliki dodatka za ogrodje *Etage* za podatkovno pretokovno delo z grafi in ga prav tako ponudili[1] pod licenco GPL.

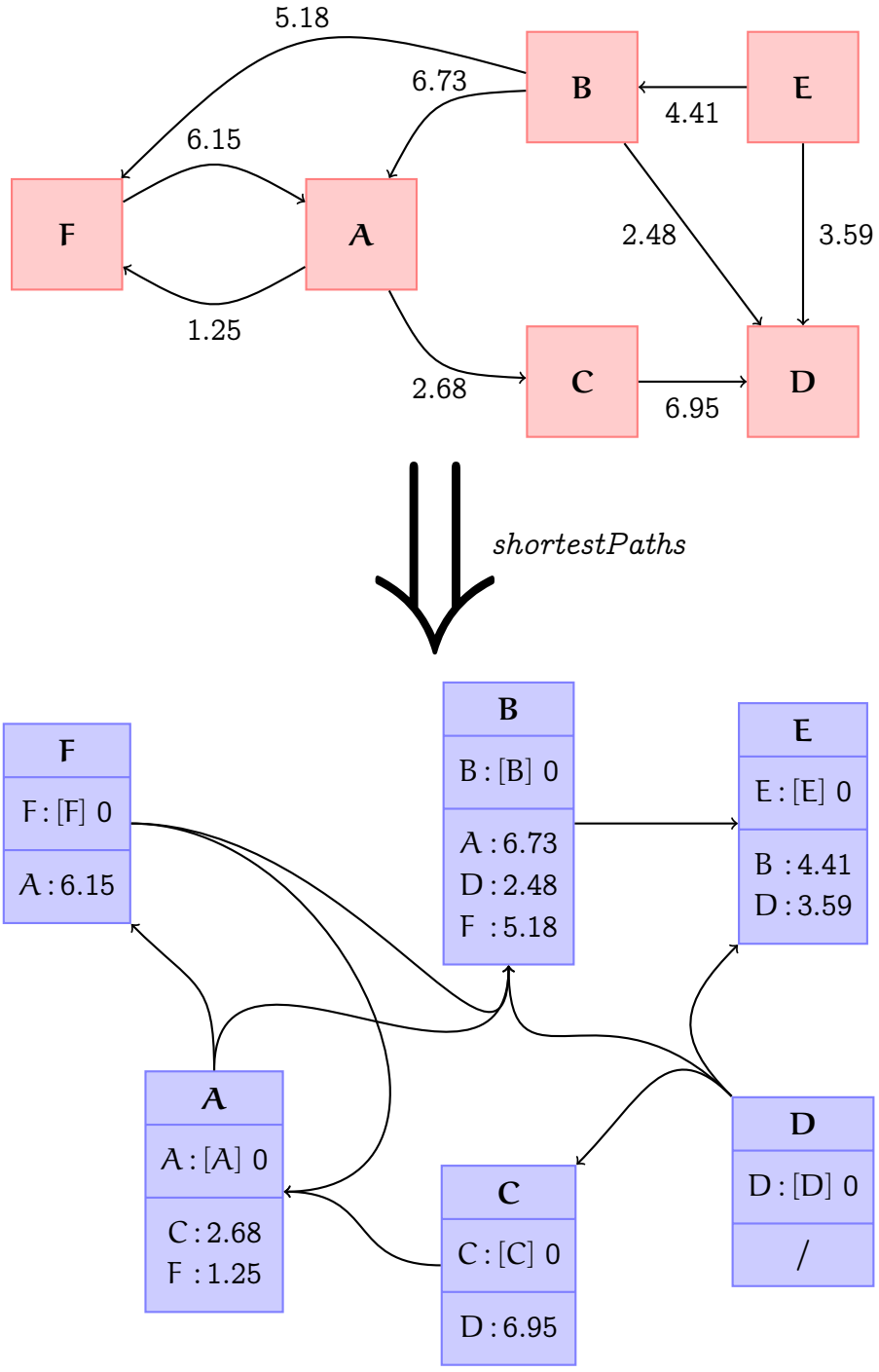
4.0.6 Opis delovanja

Na začetku preslikamo graf v mrežo operacij tipa *NodeNeuron* in povezav med njimi tako, da vozlišča preslikamo v operacije, povezave pa povežemo v obratni smeri, kot so v grafu. V vsako vozlišče/operacijo shranimo še podatke o ceni izhodnih povezav v podanem grafu. Primer takšne preslikave, ki je v kodi v dodatku A definirana v funkciji *shortestPaths*, lahko vidimo na sliki 4.1. Funkcija *shortestPaths* je tudi primer definicije funkcije, ki abstrahira delovanje mreže v funkcijo, tako da se lahko potem modularno uporabi kot podmreža znotraj večje mreže.

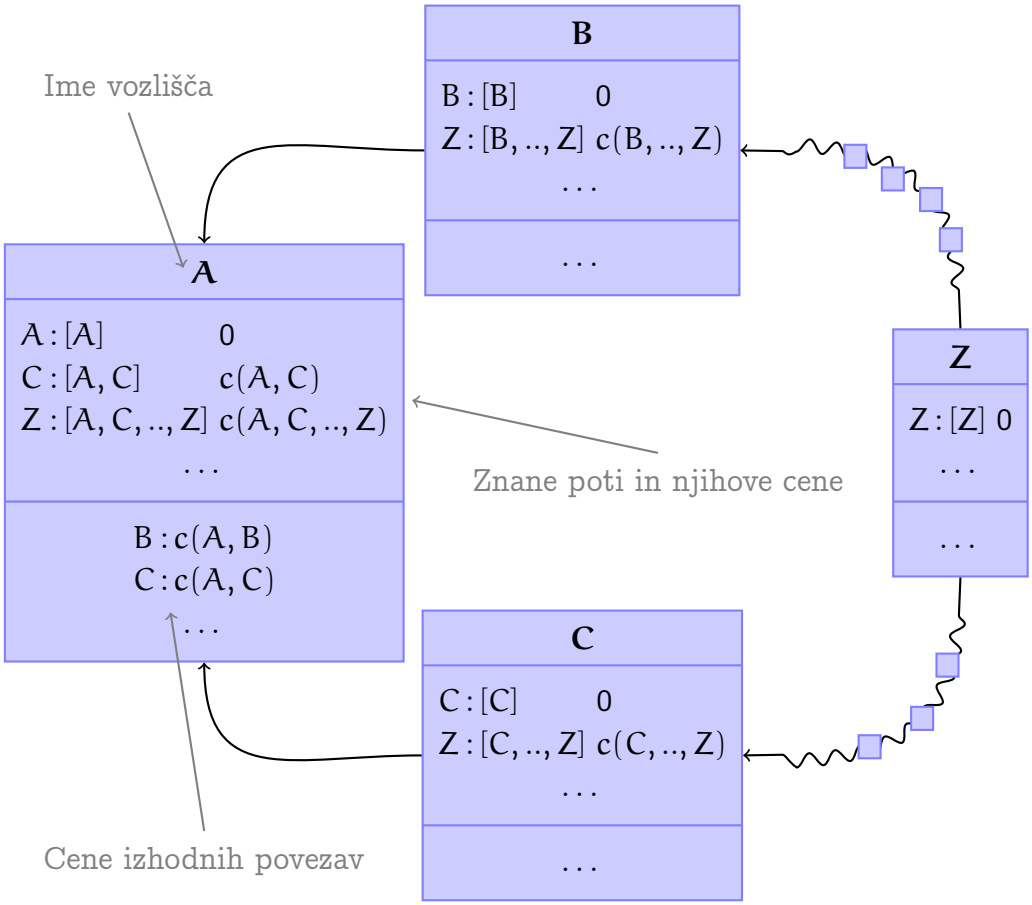
Po izgradnji mreže sprožimo računanje s funkcijo *sendTopologyChange*, ki obvesti vsa vozlišča/operacije, da pošljejo svojim sosedom podatke o njim trenutno znanih poteh.

Za ponazoritev delovanja samega vozlišča/operacije si pomagajmo s sliko 4.2. Vsako vozlišče/operacija pomni trenutno izračunano najcenejšo pot do vsakega drugega vozlišča/operacije (urejen seznam vozlišč/operacij na poti) ter

[1] `http://hackage.haskell.org/package/Etage-Graph`



Slika 4.1: Primer preslikave grafa v mrežo operacij.



Slika 4.2: Ponazoritev delovanja računanja najcenejše poti v grafu.

seveda tudi ceno te poti (na sliki 4.2 označeno s funkcijo c). Prav tako smo ob preslikavi grafa v mrežo operacij shranili tudi cene vseh izhodnih povezav v grafu. Te izhodne povezave grafa smo v fazi preslikave preslikali v vhodne povezave mreže, saj se podatki o dosegljivih vozliščih/operacijah prenašajo nazaj po povezavah v grafu, ki delajo ta vozlišča/operacije dosegljive.

Na začetku vsako vozlišče/operacija vsebuje le pot do samega sebe z ničelno ceno. Ob spremembi topologije grafa (oziroma po začetni izgradnji mreže ob zagonu programa) vozlišča/operacije razpošljejo svojim sosedom podatke o tem, kakšne so njim znane poti in njihove cene. Ko sosednja ope-

racija sprejme podatek o poti, jo primerja z njej znano trenutno najcenejšo potjo do istega končnega vozlišča/operacije in če je ta nova pot, vključno s povezavo nazaj do pošiljatelja te poti, cenejša, posodobi pri sebi shranjeno najcenejšo pot s to novo skupno potjo in jo zatem pošlje naprej svojim sosedom, ki vse skupaj ponovijo.

Ponazorimo to s primerom. Naprimer, da je vozlišče/operacija B na sliki 4.2 poslala svojim sosedom podatek o njej znani poti do Z. Ta podatek A sprejme in primerja s potjo do Z, ki jo sama pozna (v kolikor takšne poti še ne pozna, se na to gleda kot na pot z neskončno ceno). V tem primeru tako primerja ceno $c(A, C, .., Z)$ s $c(A, B) + c(B, .., Z)$. Če je takšna nova skupna pot $[A, B, .., Z]$ boljša/cenejša, si jo A shrani in pošlje naprej svojim morebitnim sosedom.

Kodo opisanega delovanja lahko najdemo v dodatku A v funkciji *run* v delu, ki obdeluje konstruktor podatka *TopologyUpdate*.

Takšno delovanje sčasoma skonvergira k optimalnim potem, medtem pa lahko uporabnik že hitro začne dobivati informacije o trenutno najdenih poteh, kar je v določenih problemskih domenah lahko koristno. Hkrati se, po začetni vzpostavitvi mreže, ki predstavlja graf, lahko le-to spreminja in spreminja tudi cene na povezavah, pri čemer ni potrebno mreže zgraditi znova oziroma ponoviti celotnega računanja najcenejših poti, ampak lahko program uporabi že izračunane poti in jih samo po potrebi posodobi. Ta funkcionalnost posodabljanja grafa v okviru tega dela ni bila implementirana.

4.0.7 Analiza delovanja

Zaradi relativne enostavnosti primera (vse operacije istega tipa, le en tip podatkov, ki se prenašajo po povezavah, enolično delovanje) ga lahko uporabimo tudi za okvirno zmogljivostno analizo delovanja ogrodja *Etage*,

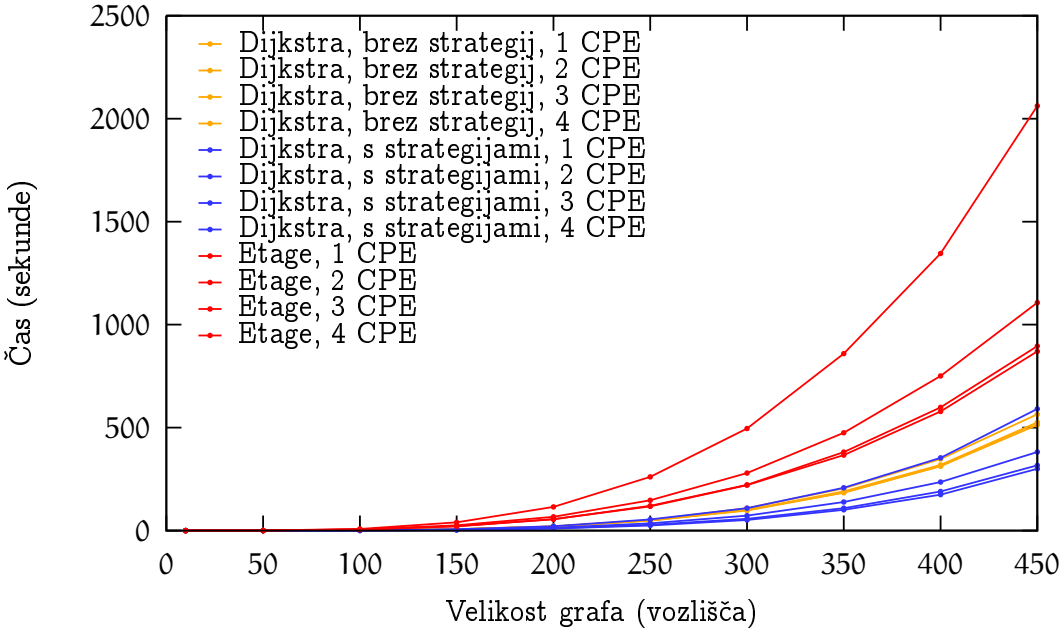
saj lahko iskanje najcenejših poti po pravkar opisanem pristopu primerjamo z naprimer Dijkstrinim algoritmom za iskanje najcenejših poti ter tako dobimo približno zmogljivostno predstavo o delovanju ogrodja.

Pri tem se je potrebno zavedati, da opisani pristop izračuna najcenejše poti med vsemi vozlišči/operacijami hkrati, tako da smo ga primerjali s poganjanjem Dijkstrinega algoritma, ki sicer izračuna le najcenejše poti iz enega vozlišča, nad vsakim vozliščem grafa. Takšen pristop je koristen tudi za to, ker ga je možno enostavno računati vzporedno, saj lahko računamo za vsako vozlišče neodvisno. Za Dijkstrin algoritem smo uporabili implementacijo Haskell knjižnice za delo nad grafi[2] [11].

Meritve smo opravili na različnih velikostih naključno zgeneriranih grafov ($|E| \approx |V|^{1.8}$) in pri tem uporabili 1, 2, 3 ali 4 centralne procesne enote (CPE), da smo lahko opazovali uspešnost vzporednega računanja. V primeru Dijkstrinega algoritma smo pogledali kako se obnese brez namiga prevajalniku o možnostih vzporednega računanja in z njim, za kar smo uporabili strategije (špekulativnega) vzporednega računanja [19, 29].

Na sliki 4.3 so prikazani rezultati meritev. Kot vidimo je uporaba ogrodja pri največjih grafih nekje za faktor 3.4 počasnejša od Dijkstrinega algoritma (s strategijami) pri eni CPE in za faktor 2.8 pri štirih. Takšen faktor pripisujemo vsemu potrebnemu dodatnemu režijskemu delu, ki je povezano s pošiljanjem podatkov po povezavah, prav tako pa so tudi uporabljene podatkovne strukture zaradi svoje kompleksnosti, neurejenosti in velikosti manj prijazne do predpomnilnika. Z večanjem števila procesorjev se ta faktor manjša, kar si razlagamo s tem, da zna ogrodje bolje izkoristiti možnosti za vzporedno računanje kot slepo vzporedno neodvisno računanje Dijkstrinega algoritma za vsako vozlišče grafa.

[2] `http://hackage.haskell.org/package/fgl`



Slika 4.3: Trajanje računanja vseh najcenejših poti v grafu za Dijkstrin algoritem in pristop na osnovi ogrodja *Etage*.
Čas znotraj ene barve z večanjem števila CPE po pričakovanjih pada.

Čeprav Haskell odkrije nekaj malega možnosti za vzporedno računanje ob večkratnem poganjanju Dijkstrinega algoritma za vsako vozlišče grafa, se komaj z uporabo namiga s strategijami (špekulativnega) računanja leta uspešno izvaja vzporedno. Na eni CPE je računanje z uporabo namiga počasnejše kot sicer, kar pripisujemo potrebnemu dodatnemu režijskemu delu za pripravo vzporednega računanja, ki pa je le pri eni CPE nekoristno.

Zmogljivostno je torej uporaba podatkovno pretokovnega ogrodja slabša kot sorodni ukazno pretokovni pristopi, kar kot že omenjeno pripisujemo predvsem potrebnemu večjemu režijskemu delu pri pošiljanju podatkov po mreži. Vpliv tega je v praksi odvisen od problemske domene, ki jo z ogrodjem rešujemo. V večini primerov težav zaradi tega ne bomo imeli, v primerih, kjer pa se to izkaže za težavo, pa lahko vedno problematičen del delovanja

implementiramo imperativno ter ga kot enotno operacijo ponudimo ogrodju.

Zanimivo bi bilo raziskati, kako bi se obnesle sestavljive pomnilniške transakcije [8, 12], ki bi se zaradi svoje optimistične narave dela s podatki lahko izkazale za zmogljivostno bolj primerne. V vsakem primeru je potrebno zmogljivostne lastnosti ogrodja bolje raziskati kot tudi odkriti morebitne načine optimizacij, ki jih v ukazno pretokovnem računanju mogoče niti ne poznamo.

Poglavje 5

Primer uporabe: krmiljenje robota

Ob samem računanju oziroma izvajanju podatkovno pretokovnih programov znotraj računalnika, lahko tok podatkov odpremo tudi navzven, z napravami zunaj računalnika, pa tudi uporabnikom, brez potrebe po vpeljavi novih konstruktov v sam model računanja in s tem tudi ogrodje. Še vedno je program sestavljen iz operacij in povezav, kjer pa so operacije oziroma viri podatkov lahko tudi druge naprave, drugi programi, lokalni ali oddaljeni, pa tudi uporabnik sam oziroma več njih. Vse to pomeni tudi lažjo in boljšo modularnost, ponovno uporabo ter razširljivost, saj ohranjamo sestavne dele programa enake in združljive.

Ta prednost je še posebej opazna ob primerjavi s pristopi uporabljenimi v imperativnih programih. Najbolj pogosto so to zanke, ki čakajo na dogodke s strani uporabnika, in delo z datotekami in napravami skozi razne vmesnike. Ob samem logičnem toku delovanja programa je pogosto težavno uspešno in učinkovito preplesti omenjene pristope. Naprimer, če poteka branje iz datoteke predolgo, se lahko zgodi, da kakšen uporabnikov dogodek (pre)pozno obdelamo. Poleg tega je tudi sama koda nepregledna, saj mora prepletati več tokov operacij med seboj. Kot rešitev se pogosto uporabljajo

niti, vendar to vpelje še dodatno komponento v programiranje, ki se velikokrat izkaže za težavno zaradi potrebe po ustrezni sinhronizaciji med ločenimi tokovi operacij.

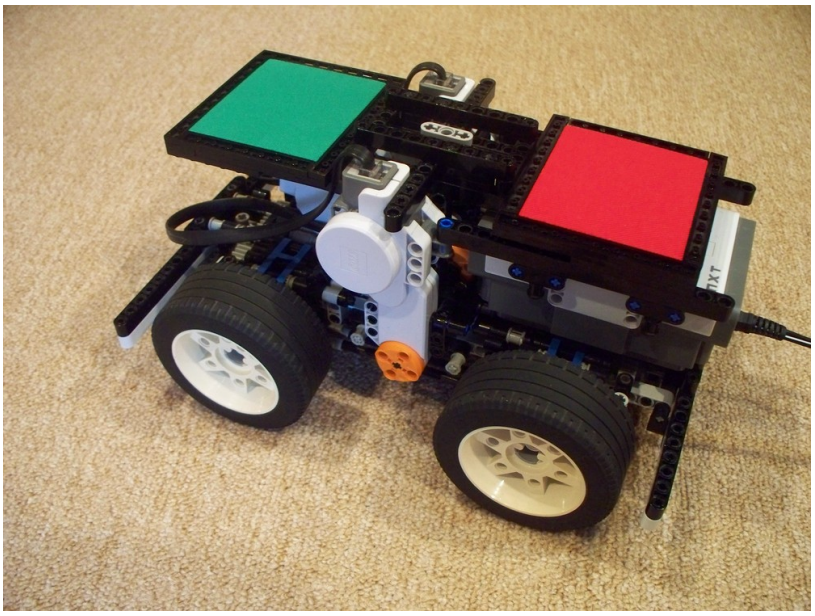
Prav zato je dobro razširiti osnovni model podatkovno pretokovnega računanja v model dogodkovnih mrež, torej dogodkovno pretokovno računanje, kjer je torej za vsak podatek znan tudi njegov čas nastanka. To je pogosto uporabno tudi pri podatkovno pretokovnem računanju, naprimer, za zaznavanje nezaželjenih ciklov povezav ali pravilen odziv operacij na zakasnele oziroma neaktualne podatke. Koristno je tudi pri podatkih, ki prihajajo iz različnih naprav, morebiti tudi oddaljenih med seboj. Ker v našem ogrodju vsi podatki vsebujejo podatek o času svojega nastanka, razlikovanja med podatki in dogodki ne delamo in je prepuščeno konkretni uporabi ogrodja, da uporabi ali ne uporabi tudi dogodkovno naravo podatkov. Zato tudi v tem delu ne ločujemo med njimi in modeli na njihovi osnovi.

5.1 Zasnova robota

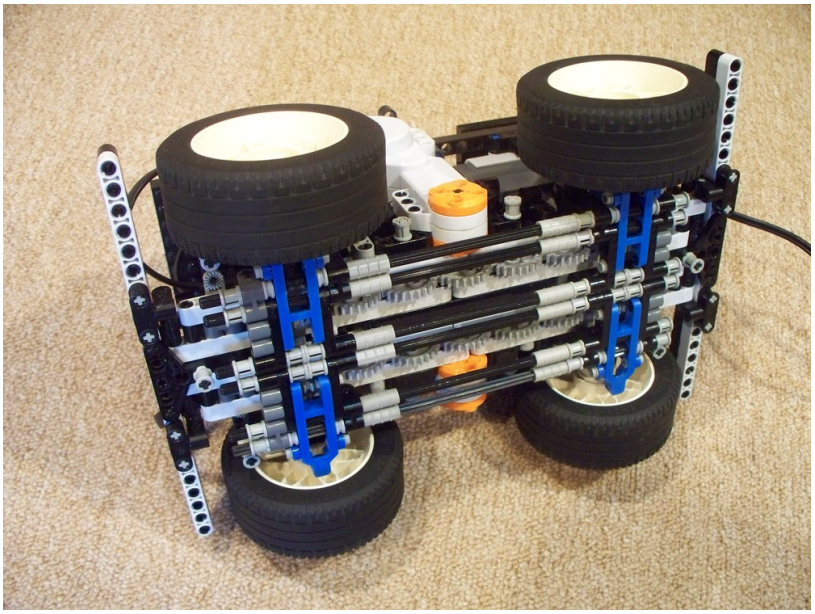
Kot smo že predstavili v poglavju 2, smo robota zgradili z uporabo Lego kock na osnovi kompleta Lego Mindstorms NXT. Da bi bilo samo upravljanje robota natančno, stabilno in brez podrsavanj koles, smo se po nekaj iteracijah gradnje robota odločili za model s pogonom na štiri kolesa in možnostjo zavijanja z zavijanjem koles, predstavljen na slikah 5.1, 5.2 in 5.3.

Na vrhu slike 5.1 so vidni markerji za pomoč pri določanju položaja robota s pomočjo kamere. Spredaj se vidi možnost priklopa napajanja.

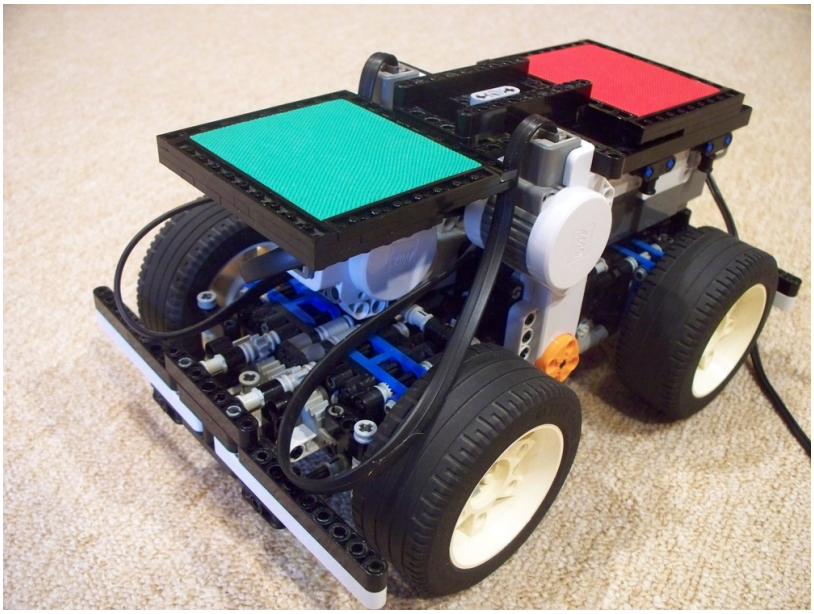
Slika 5.2 prikazuje, kako vsak motor preko vrste zobnikov poganja svoj par koles. Motorja sta nameščena v sredini, da so razdalje med motorji in kolesi enake, tako da se enakomerno in enako odzivajo na delovanje motorjev.



Slika 5.1: Robot na štirikolesni pogon z motorjema na vsaki strani.



Slika 5.2: Pogled od spodaj.



Slika 5.3: Pogled od zadaj.

Na koncu se s krogličnim ležajem, ki omogoča zavijanje koles, prenese vrtenje na kolesi. Vsak par koles na vsaki strani krmili po en motor, tretji motor pa skrbi za zavijanje koles.

Tretji motor, ki skrbi za zavijanje koles, je prikazan na sliki 5.3. Ta slika prav tako kaže tudi prostor, ki je na voljo za morebitne dodatne senzorje (naprimer kompas, pospeškometer ...).

Slabost predlaganega modela je, da je pred uporabo potrebno umerjanje zavijanja koles na ničeln položaj, kar pa je hkrati tudi mehanizem popravljanja različnega delovanja stranskih motorjev, ki se tako kompenzira z ničelnim položajem, ki upošteva to različno delovanje.

Model robota je pogojen predvsem z našimi željami in potrebami po motoriki robota in senzorjih, medtem ko resnični krmilni del robota, krmilni program na osnovni našega ogrodja *Etage*, teče neodvisno od robota na računalniku, s katerim komunicira preko brezžične povezave Bluetooth.

Zaradi poenostavitve primera smo se omejili le na en senzorični podatek o robotu in to je lokacija in usmerjenosti robota v poligonu, kar smo pridobivali preko podatkov iz kamere, in nismo uporabili morebitnih senzorjev na samem robotu.

5.2 Določanje položaja robota

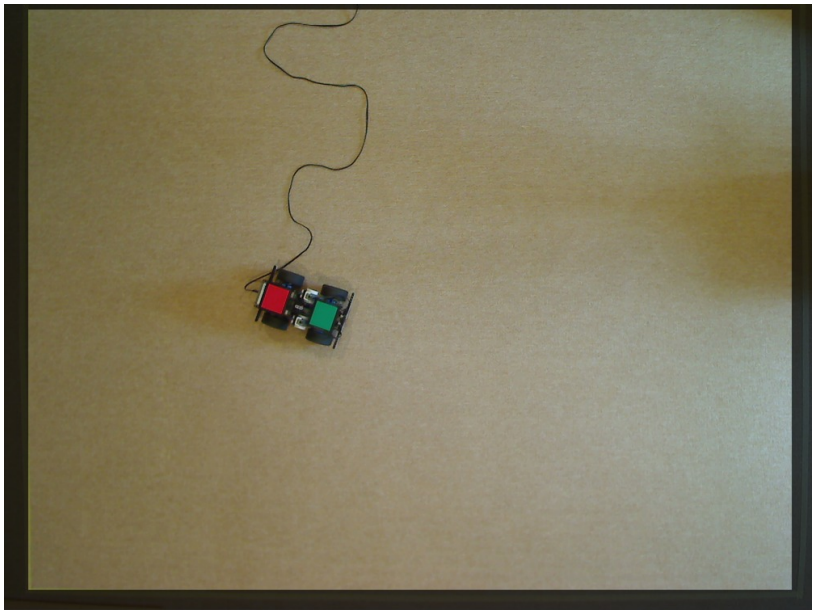
Za določanje položaja robota smo uporabili spletno kamero Logitech HD Pro Webcam C910, nameščeno nad poligonom (slika 5.4), ki zajema sliko tlorisa poligona (slika 5.5). Zaradi lažjega določanja položaja robota je ta opremljen z barvnimi markerji (slika 5.1), tako da je obdelava slike iz kamere lažja, saj je potrebno prepoznati le barvne markerje in na podlagi njihovega položaja na sliki izračunati položaj robota v prostoru. S kamero smo zajemali 15 barvnih sličic na sekundo (FPS) z ločljivostjo 1600x1200 slikovnih pik.

Na sliki 5.4 je tudi spletna kamera Logitech QuickCam Pro 9000, ki smo jo uporabljali vzporedno za snemanje samega dogajanja v poligonu.

Kot že omenjeno, se določanje položaja robota v prostoru dobi iz položaja enostavno prepoznavnih markerjev na sliki kamere. Zaradi čimlažje prepoznave so enobarvni, da pa je možno določiti usmerjenost robota, sta potrebna dva markerja. Izbrali smo rdečo in zeleno barvo, saj imata po barvni teoriji največjo intenziteto in s tem najbolj izstopata v HSV (hue-saturation-value) barvnem prostoru od drugih barv.

Sam izbor materiala iz katerega sta markerja se je tudi pokazal kot zanimiv problem, saj je zaželjeno, da je po eni strani čimbolj barven, čimbolj svetel (čimbolj odbija barvo), hkrati pa ne sme biti lesketajoč, saj to povzroči, da se na določenih položajih markerjev zaslepi kamera. Ob tem je koristno tudi, če se ne pojavljajo sence oziroma se razpršijo in nimajo ostre

Slika 5.4: Spletni kameri Logitech HD Pro Webcam C910 in Logitech QuickCam Pro 9000 pritrjeni nad poligonom.



Slika 5.5: Slika poligona, kakor jo vidi kamera. Nastavljivo je področje, v katerem se pričakuje robot, kar je prikazano s potemnjenim robom.

Slika 5.6: Poligon. Pri stropu je kamera in nekaj reflektorjev. Na tleh oznaka vidnega polja kamere.

oblike. Preizkusili smo markerje iz Lego kock, ki imajo sicer jasno in kričečo barvo, a so pod določenim kotom lesketajoči, tako da ima potem določanje položaja z njimi slepe pege. Barvni papir se je izkazal za povprečnega, saj ni tako odporen na sence. Najbolje se je izkazalo grobo a barvno močno blago, ki ima odlično enakomerno barvo in nobenih senc, saj jo zaradi svoje strukture razprši, sama struktura pa se še vseeno ne vidi na kameri. Takšna

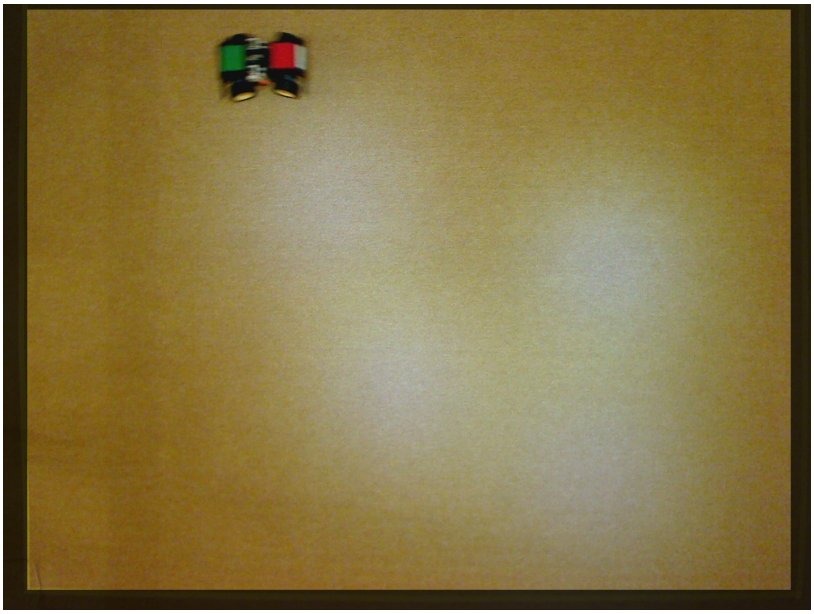
markerja sta vidna tudi na slikah 5.1, 5.2 in 5.5.

5.2.1 Izračun položaja v prostoru glede na sliko

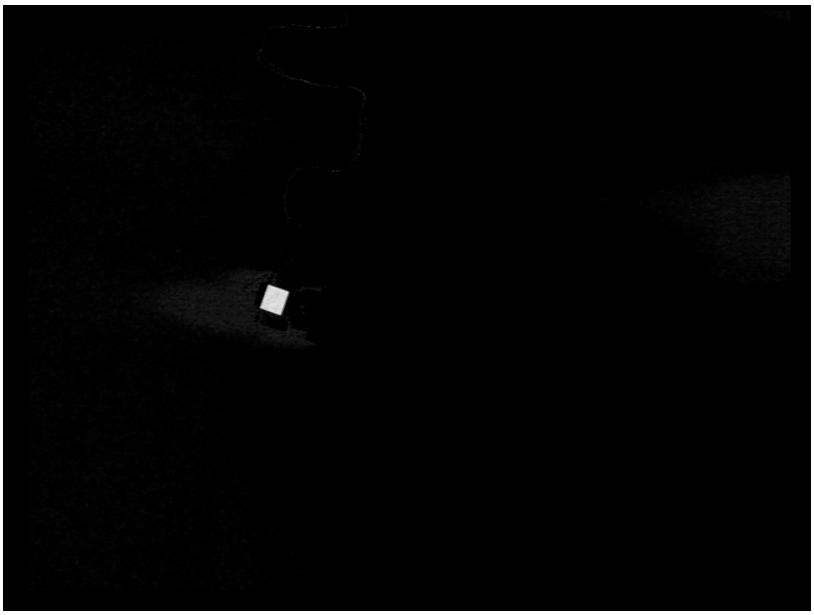
V ta namen smo razvili lasten program za zajem položaja robota v prostoru, saj so se nekateri obstoječi, ki smo jih preizkusili, izkazali za neustrezne. Program mora biti sposoben obdelati tudi slike z robotom v gibanju in tudi na podlagi teh izračunati položaj robota. Primer takšne slike je 5.7.

Zaradi potrebe po hitrem delovanju je program implementiran v programskem jeziku C. Prepoznane položaje lahko sporoča kar preko standardnega izhoda, kar omogoča modularno zasnovo celotnega sistema, kakor s preusmeritvijo (naprimer preko povezave SSH) tudi delovanje preko mreže, brez nepotrebnih dodatnih protokolov. Smo pa za večjo odzivnost razvili tudi direktno integracijo v ogrodje v primeru lokalnega delovanja na istem računalniku.

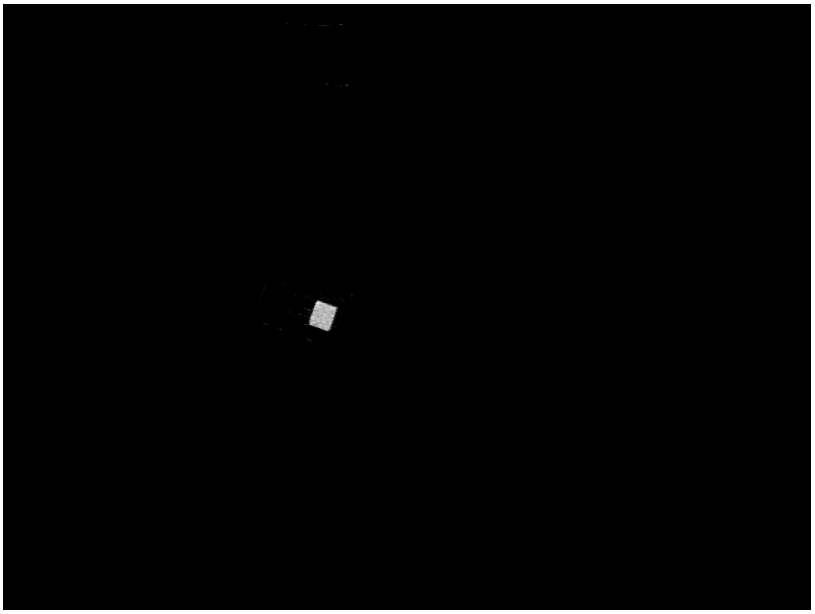
Vsako sliko v prvem koraku predobdela tako, da izpostavi željene barve, v našem primeru rdečo in zeleno, kot prikazano na primeru na slikama 5.8 in 5.9, ki jih potem lahko enostavno najde in statistično izračuna središče markerjev ter s tem lokacijo in usmerjenost robota v poligonu. Zaradi delovanja, ki temelji le na barvah in statističnih lastnostih markerjev (kompaktnosti oblike in enotnosti barve) deluje tudi v primeru "razmazanih" markerjev in zaradi prvega koraka, ki predobdela sliko glede na barve, deluje tudi v slabših svetlobnih pogojih. Zaradi statistične obdelave markerjev je tako izračunan položaj markerja na sliki odporen na šum pri zajemu slike, hkrati pa se doseže večjo od slikovne pike natančnost, kar omogoča določanje položaja v prostoru tudi na milimeter natančno. Sicer je absolutna natančnost potem v praksi manjša, zaradi kopičenja napak še iz drugih virov (naprimer kalibracije same kamere).



Slika 5.7: "Razmazani" markerji zaradi gibanja robota, ki nimajo jasne meje. Hkrati se vidi tudi vpliv perspektive, saj so markerji precej zamaknjeni glede na sredino robota glede na ravnino poligona.



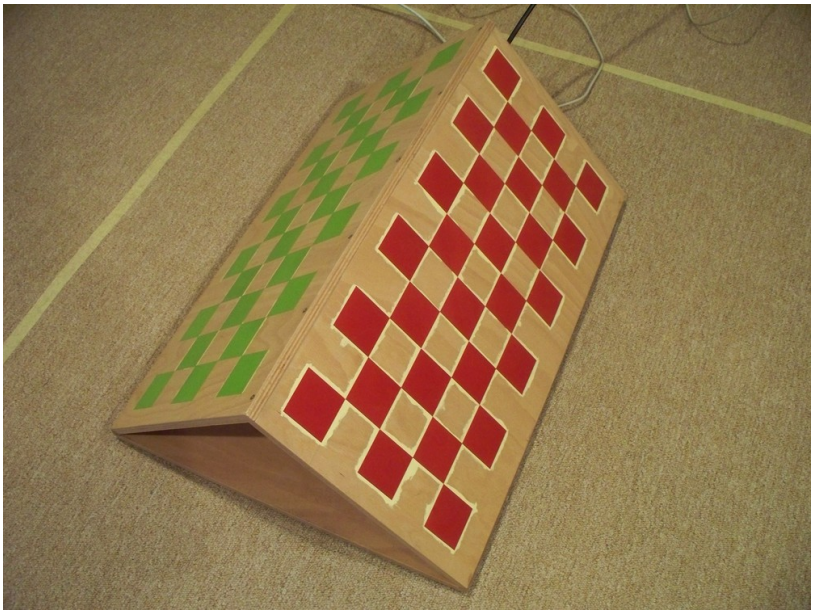
Slika 5.8: Predobdelava slike 5.5, ki izpostavi rdeče markerje.



Slika 5.9: Predobdelava slike 5.5, ki izpostavi zelene markerje.

Ko se določi položaj markerja na sliki, ga je potrebno preslikati v koordinate prostora poligona. V splošnem želimo enotno ravnino v kateri računamo položaje objektov, še posebej, če uporabljamo različne objekte različnih višin (robota in ovir naprimer). Zato je potrebno kamero oziroma to preslikavo/transformacijo skalibrirati v tridimenzionalnem prostoru. Izkaže se namreč, da je pri uporabi kamere nad poligonom vpliv perspektive na robovih poligona precejšnji in s tem tudi napaka, če ne upoštevamo pravilno različnih višin objektov. V primeru poligona, ki smo ga uporabljali, in robota, ki ima markerje na višini 12,9 cm, je na robu poligona ta razlika že več kot 15 cm v primerjavi s tlemi, kar se lahko vidi tudi na sliki 5.7.

Postopek za kalibracijo kamere smo implementiral po postopku opisanem v [30]. Za kalibracijo se pod kamero postavi tridimenzionalni objekt znane oblike in velikosti, najboljše kar z markerji razporejenimi v obliki šahovnice, saj je tako najlažje pridobiti znane koordinate na tem objektu. V ta na-



Slika 5.10: Kalibracijski objekt za tridimenzionalno kalibracijo kamere.

men smo izdelali ustrezen kalibracijski objekt iz lesa prikazan na sliki 5.10: tristrano prizmo, kjer je na vsaki od zgornjih dveh ploskvah šahovnica v različnih barvah (rdeči in zeleni) velikosti 11x5 kvadratkov, kar da 10x4 notranjih oglišč, ki so primerni za enostavno določanje na sliki, prav tako pa poznamo njihove koordinate v prostoru.

Objekt je izdelan večnamensko, tako da ob svoji kalibracijski vlogi vsebuje na eni strani krmilni računalnik, slika 5.11, na drugi strani pa prostor v katerega se lahko parkira robot. Zaradi natančnega znanega položaja tega prostora se lahko tako uporabi za naprimer učenje robota, da se vrne "domov".

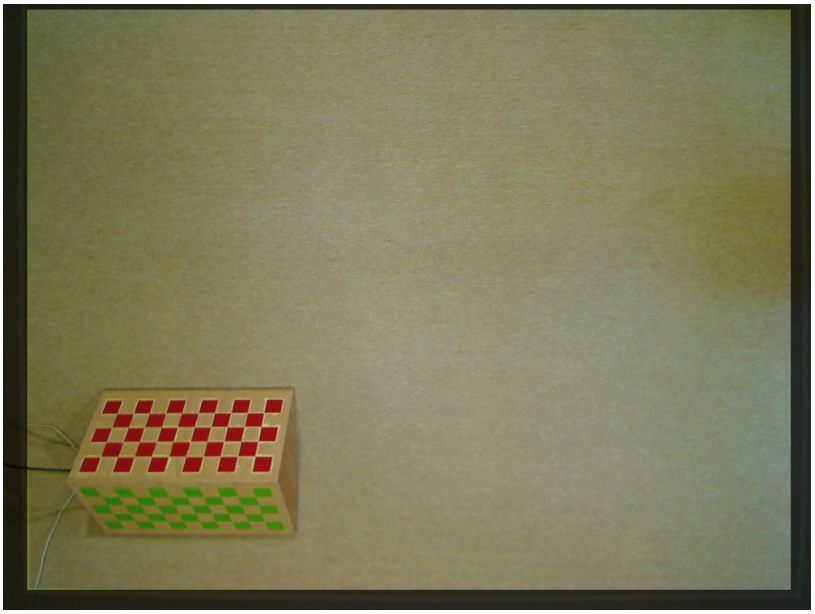
Na sliki poligona, primer na sliki 5.12, s postavljenim kalibracijskim objektom določimo koordinate oglišč šahovnic, kot je prikazano na sliki 5.15. Za ta namen smo uporabil algoritem knjižnice OpenCV[1]. Podobno kot pri

[1] `http://opencv.willowgarage.com/wiki/`

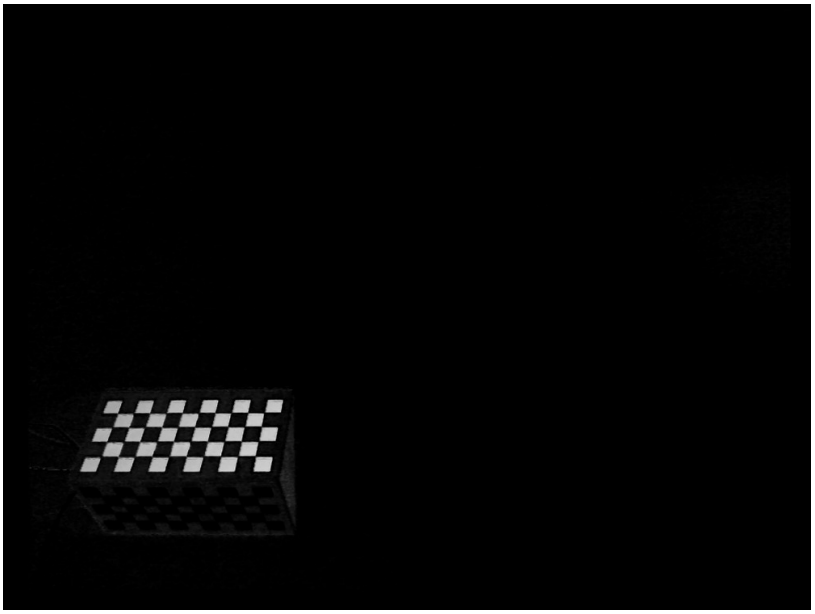
Slika 5.11: Kalibracijski objekt vsebuje na eni strani prazen prostor, v katerega se lahko parkira robot. Na tej strani pa vsebuje krmilni računalnik.

markerjih robota se tu slika predobdela za vsako posebej, tako da se le-ta izpostavi, kot je prikazano na slikama 5.13 in 5.14. Zatem se posreduje knjižnici OpenCV, ki najde oglišča šahovnice. Tako pridobljene koordinate oglišč na sliki (oziroma še boljše, iz več slik poligona) se skupaj z njihovimi znanimi koordinatami v prostoru sestavijo v projekcijsko matriko, za katero se najde rešitev, oziroma zaradi netočnih vrednosti (in veliko večjega števila enačb, kot je neznank) se najde rešitev po metodi najmanjših kvadratov. Za ta namen se uporabi SVD razcep matrike in se kot rešitev izbere lastni vektor, ki ustreza najmanjši lastni vrednosti.

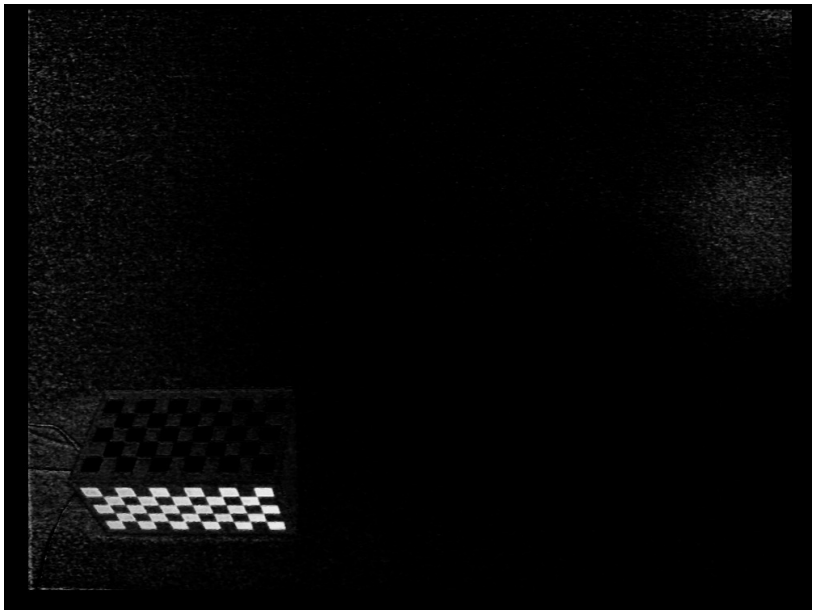
Tako pridobljena rešitev projekcijske matrike se kasneje uporablja za preslikavo/transformacijo iz znanih koordinat na sliki v koordinate v prostoru poligona.



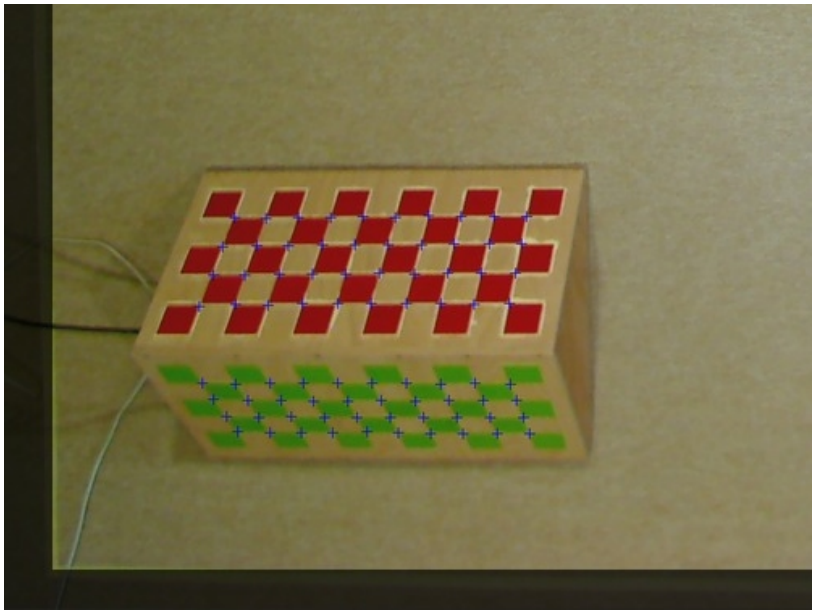
Slika 5.12: Slika poligona s kalibracijskim objektom, kakor jo vidi kamera.



Slika 5.13: Predobdelava slike kalibracijskega objekta s slike 5.12, ki izpostavi rdeče markerje šahovnice.



Slika 5.14: Predobdelava slike kalibracijskega objekta s slike 5.12, ki izpostavi zelene markerje šahovnice.



Slika 5.15: Najdena oglišča obeh šahovnic kalibracijskega objekta, označeni z modrimi križci. [izsek]

5.2.2 Implementacija

Glavno vodilo pri implementaciji določanja položaja robota je bila hitra odzivnost oziroma bolj natančno čimkrajši čas med za zajemom slike iz kamere in njegovo pojavitvijo kot dogodka v mreži ogrodja. Saj se lahko le tako robot pravočasno odziva na svoje okolje. Prva zakasnitev pride že zaradi diskretnega vzorčenja slike kamere. Pri frekvenci 15 sličic na sekundo to pomeni 67 milisekund povprečne zakasnitve, preden sploh kamera pošlje sliko računalniku po vodilu USB. Nato sledi obdelava v jedru operacijskega sistema, prenos med uporabljenimi knjižnicami kjer se zgodi tudi dekodiranje kompresiranega formata MJPEG (ki se uporablja, da je prenos do računalnika hitrejši oziroma zadosti hiter), potem iskanje markerjev v sliki in izračun položaja, prenos v Haskell in končno v samo mrežo ogrodja.

Obdelave v samem jedru ni veliko, predvsem se le abstrahira dostop do kamere v vmesnik Video4Linux (V4L). Morali pa smo popraviti manjšo napako v delovanju, da sta izbrani kameri pravilno delovali, kar smo tudi posredovali razvijalcem Linux jedra[2] za vključitev v nadaljnje verzije jedra.

Hiter prenos med knjižnicami lahko dosežemo tako, da zagotovimo direktno dostopanje knjižnic do pomnilnika z uporabo `mmap` sistemskega klica in ob tem še preprečimo morebitno odvečno kopiranje pomnilnika, za kar smo morali poskrbeti predvsem v knjižnici OpenCV[3].

Izkazalo se je tudi, da knjižnica `libv4l`, ki jo uporablja OpenCV za dekodiranje slik, uporablja izjemno počasno dekodiranje formata MJPEG, ki je bilo celo tako počasno, da pri tako visoki ločljivosti in frekvenci sličic, računalnik ni zmogel sproti dekodirati prihajajočih slik. Zato smo knjižnico `libv4l`[4] prilagodili tako, da je uporabljala dekodiranje knjižnice oziroma dru-

[2] `http://www.mail-archive.com/linux-media@vger.kernel.org/msg13406.html`
[3] `https://code.ros.org/trac/opencv/ticket/632`
[4] `http://freshmeat.net/projects/libv4l`

žine knjižnic FFmpeg[5], znane po svojih hitrosti. Razlika je bila precejšnja. Medtem, ko je prej knjižnica `libv4l` porabila v povprečju 0,0257 milisekunde na slikovno piko, je po novem porabila le 0,0087 milisekunde. Pri testni resoluciji 2592x1944 to pomeni razliko med 130 milisekundami pred v primerjavi s 44 milisekundami po spremembi, v povprečju. Torej le največ 7 sličic na sekundo prej v primerjavi s 23 sličicami potem. Pri uporabljeni resoluciji 1600x1200 to pomeni le okoli 17 milisekund zakasnitve zaradi dekompresije. Tudi ta popravek smo posredovali razvijalcem[6].

Iskanje markerjev v sliki lahko pohitrimo z raznimi hevristikami, kot naprimer iskanje v spirali okoli predhodno znanega položaja markerja, saj se robot v splošnem ne more premakniti poljubno daleč med dvema slikama kamere. Prav tako lahko iščemo s korakom, kjer upoštevamo najmanjšo pričakovano velikost markerja na sliki.

Pohitritev prenosa v Haskell smo izvedli tako, da smo sam program za izračun položaja robota neposredno vključili v ogrodje, tako da so vrednosti položaja robota na voljo kar kot spremenljivke Haskell programa. Za tem je potrebno le še te vrednosti ustrezno zapakirati v dogodek ter ga poslati po mreži.

Celotna ena meritev oziroma izračun položaja robota v prostoru z zajemom slike, obdelavo in izračunom do vstopa v mrežo tako na uporabljenem krmilnem računalniku traja v povprečju 21 milisekund. K temu je potrebno prišteti še povprečno zakasnitev zaradi samega vzorčenja pri 15 sličicah na sekundo, da dobimo povprečno zakasnitev med dogodkom na poligonu in vstopom le-tega v mrežo, 88 milisekund.

Pri kalibraciji se je iskanje šahovnice s pomočjo knjižnice OpenCV izkazalo za premalo robustno pri resni uporabi v različnih svetlobnih pogojih in

[5] `http://www.ffmpeg.org/`

[6] `http://www.mail-archive.com/linux-media@vger.kernel.org/msg23783.html`

pod različnimi koti šahovnice. Zato smo algoritem za iskanje šahovnice in njenih oglišč izboljšali ter posredovali razvijalcem[7].

5.3 Upravljanje z robotom

Če želimo našo mrežo uspešno razširiti z napravami zunaj računalnika, moramo poskrbeti, da je sam prenos podatkov med temi napravami transparenten in da z vidika ogrodja ni opazen. Ker pa smo definirali delovanje našega ogrodja kot v osnovi nedeterministično, lahko dovolimo zakasnitve pri pošiljanju podatkov po povezavah (ki v tem primeru predstavljajo realne povezave med napravami), saj jih lahko prištevamo ravno tej nedeterminističnosti. Po drugi strani pa še vedno ne smemo dovoljevati izgube podatkov ob pošiljanju po povezavah, kar bi se prav tako lahko dogajalo na povezavah med napravami.

Seveda v praksi ta teoretična možnost dovoljene poljubne zakasnitve podatkov vseeno predstavlja oviro pri normalnem delovanju mreže, še posebej, če želimo z njo krmiliti robota v realnem času. Zato je potrebno poskrbeti, da so te zakasnitve čimkrajše (glede na dogodke v fizičnem svetu) in da so v predvidljivih časovnih mejah. Za senzorične podatke (podatke ki prihajajo od robota) smo predstavili pristope k temu v predhodnem poglavju 5.2. Za motorične podatke (podatke ki se jih pošilja robotu) z namenom upravljanja gibanja robota pa jih bomo predstavili v nadaljevanju.

Možnosti upravljanja robota so seveda odvisne od samega robota, ki ga imamo na voljo. V našem primeru je ta zgrajen na osnovi kompleta Lego Mindstorms NXT in uporablja njeno brezžično povezavo Bluetooth. Ta omogoča v grobem dva načina upravljanja robota: direktno upravljanje robota

[7] `https://code.ros.org/trac/opencv/ticket/545`

z osnovnimi ukazi, ki so na voljo s strani proizvajalca kompleta, in posredno upravljanje z vmesnim programom, ki teče na robotu in interpretira podatke/ukaze, ki mu jih pošiljamo na način, na katerega ga sprogramiramo, ter tako upravlja z robotom. Oba načina bomo primerjali v nadaljevanju.

Za namen komunikacije med krmilnim računalnikom in robotom smo razvili ustrezno knjižnico za programski jezik Haskell. Protokol komunikacije je odprt in na voljo, tako da je v splošnem implementacija direktna, žal pa je na določenih mestih nejasna, tako da je potrebno nekatere aspekte delovanje razbrati iz kode samega operacijskega sistema krmilne enote NXT. Protokol proizvajalca podpira izvajanje direktnih ukazov za upravljanje z motorji in senzorji, kot nekaj dodatnih splošnih ukazov za dostop do preslikanih pomnilniških lokacij za še bolj neposreden dostop do krmiljenja.

Protokol proizvajalca in tudi naša knjižnica tako podpirata pester nabor možnosti upravljanja, prav tako pa je protokol razširljiv in lahko podpira naprimer tudi dodatne (neuradne) senzorje, kot je naprimer kompas, za katerega smo tudi razvili vmesnik. Po drugi strani pa so nekatere funkcionalnosti sicer definirane v protokolu, ampak niso implementirane v samem sistemu. Primer tega je možnost dostopa do informacije o kvaliteti povezave Bluetooth, ki bi lahko služila robotu tudi kot informacija o oddaljenosti od krmilnega računalnika in s tem prostora za parkiranje. To pomanjkljivost smo odpravili tako, da se kvaliteta povezave pridobi iz strani krmilnega računalnika.

5.3.1 Direktno upravljanje

Izvajanje direktnih ukazov omogoča razvoj v visoko nivojskih programskih jezikih na bolj zmogljivih računalnikih, medtem ko se sami ukazi posredujejo krmilni enoti v izvajanje.

Žal se izkaže, da je takšna uporaba direktnih ukazov pomanjkljiva v primerih, ko želimo hitro odzivnost robota na ukaze. Implementacija protokola Bluetooth je sicer v2.0 EDR (angl. enhanced data rate), ki v praksi omogoča 2.1 mbit/s prenos podatkov, ampak zamenjava smeri komunikacije že po specifikaciji implementacije traja 30 milisekund, v praksi pa tudi več. Tako pošiljanje ukazov za branje podatkov senzorjev in potem na podlagi njih ponovno ukazov za krmiljenje motorjev, zahteva kar nekaj sprememb spremeni komunikacije in s tem drastično podaljša čas izvajanja celotnega sklopa ukazov.

Tako so direktni ukazi primerni res le za zelo osnovne operacije, takoj ko pa je potrebno izvajati ukaze na podlagi podatkov senzorjev ali drugih notranjih podatkov o stanju robota, pa postanejo prepočasne. Režave se pojavijo že, če želimo hkrati pognati dva motorja (naprimer levega in desnega na robotu). Ker moramo za to poslati dva zaporedna ukaza se prvi že začne izvajati, medtem ko se drugi komaj pošilja, s čimer pride do neusklajenosti med motorjema.

5.3.2 Posredno upravljanje z vmesnim programom

Knjižnico smo po vzoru podobne knjižnice za okolje MATLAB[8] nadgradil z dodatnim programom, ki teče na krmilni enoti in omogoča samostojno izvajanje bolj kompleksnih ukazov, medtem ko se iz krmilnega računalnika pošiljajo le parametri tem ukazom. Ker žal program knjižnice za MATLAB MotorControl[9] ne podpira vsega potrebnega, čeprav ima po drugi strani nekaj odličnih funkcionalnosti, predvsem natančno delovanje, smo naredili podoben osnoven program v programskem jeziku Not eXactly C (NXC)[10], ki

[8] http://www.mindstorms.rwth-aachen.de/trac/
[9] http://www.mindstorms.rwth-aachen.de/trac/wiki/MotorControl
[10] http://bricxcc.sourceforge.net/nbc/

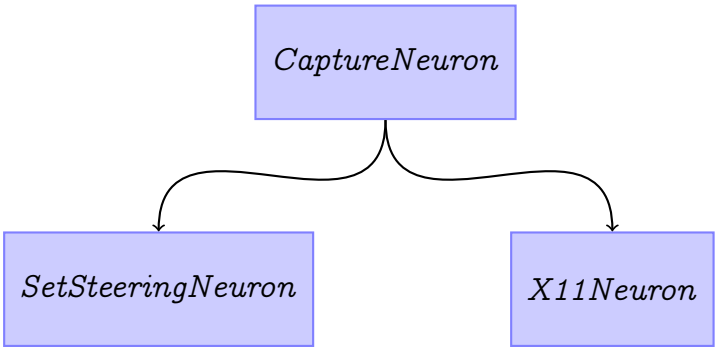
omogoča obračanje motorjev za določeno vrednost obratov ali na absolutno vrednost položaja motorja. Tudi za več motorjev hkrati. Mogoče se lahko kasneje ta funkcionalnost združi s programom MotorControl, s katerim si deli združljiv protokol pošiljanja parametrov ukazom.

Funkcionalnost programa je zasnovana tako, da je potrebno pošiljati podatke le v eno smer, od krmilnega računalnika do krmilne enote, tako da med delovanjem robota ni zakasnitev zaradi spreminjanja smeri komunikacije. Hkrati je program narejen tako, da se lahko trenutni ukaz, čeprav je še v izvajanju, prekine z novim, ne da bi bilo potrebno čakati (in torej preverjati), če se je ukaz že izvršil. Takšno pošiljanje je izvedljivo tudi zato, ker se v našem primeru uporablja le neodvisni senzor (spletna kamera) in se ne uporabljajo drugi senzorji na robotu samem. Program se tako naloži na krmilno enoto, se zažene in potem izvaja ukaze, ki mu jih pošilja krmilni računalnik. Vse to se lahko naredi kar preko povezave Bluetooth.

5.4 Umerjanje zavijanja

Možnost zavijanja robota z zavijanjem koles pomeni, da je potrebno na začetku uporabe robota umeriti motor in najti njegov ničelni položaj, pri katerem se robot premika v ravni črti. S tem se poenoti njegov interni koordinatni sistem in se omogoči, da se podatki zbrani v eni vožnji uporabijo tudi v drugih. To je še toliko bolj pomembno, če želimo učiti in testirati robota v več vožnjah. V ta namen smo razvili program, seveda na osnovi ogrodja *Etage*, ki s premikanjem robota in slednjem njegovim realnim premikom v poligonu izračuna potrebne popravke, da se robot začne premikati v ravni črti.

Na sliki 5.16 je predstavljena mreža programa za umerjanje zavijanja ro-

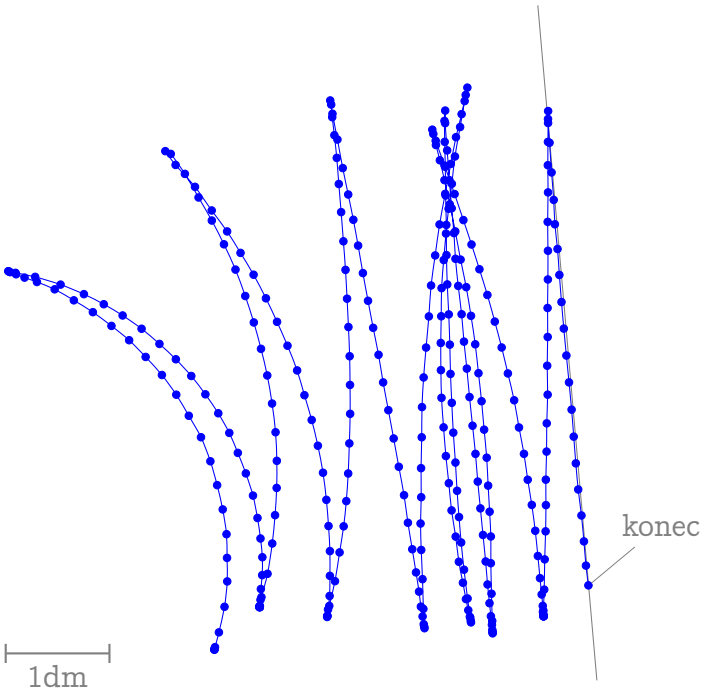


Slika 5.16: Mreža programa za umerjanje zavijanja robota.

bota. V operaciji *CaptureNeuron* se izračunajo in obdelajo podatki položaja robota, od koder se pošljejo naprej v dve operaciji, *SetSteeringNeuron* in *X11Neuron*. Slednja operacija v tem programu služi le za grafičen prikaz delovanja programa uporabniku, torej trenutnega položaja robota in njegovih premikov. V splošnem pa *X11Neuron* služi za celotno interakcijo z uporabnikom, kjer mu po eni strani grafično prikazuje dobljene podatke o robotu, po drugi strani pa od njega sprejema morebitne ukaze robotu.

Operacija *SetSteeringNeuron* v tem programu vsebuje njegovo glavno funkcionalnost, kjer se glede na pridobljene predhodne podatke o položaju robota odloča o potrebnih popravkih zavoja koles. Deluje tako, da premakne robota za določeno razdaljo naprej in ga potem vrne nazaj ter iz vseh pridobljenih vmesnih položajev izračuna trenutni kot gibanja robota. V kolikor ta ni manj kot 0.05 stopinje različen od premika v ravni črti, izračuna in ustrezno popravi zavoj koles. Za tem meritev ponovi.

Na začetku smo nameravali uporabiti kontroler PID (angl. proportional-integral-derivative), saj smo se bali, da bi lahko umerjanje brez njega ves čas nihalo okoli ničelnega položaja in se ne bi nikoli zaključilo. Izkazalo se je, da lahko z uporabo povprečne informacije vseh vmesnih položajev pridobimo meritev kota napake dovolj natančno in jo tudi dovolj natančno preslikamo

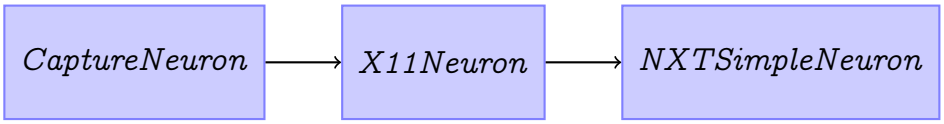


Slika 5.17: Primer poti robota pri umerjanju.
Točke predstavljajo izmerjene položaje robota.

v premik motorja, da se to ne zgodi. Ničelni položaj motorja se hitro znajde znotraj mej izbrane tolerance. Primer poti umerjanja robota z na začetku skrajno zamaknjenim ničelnim položajem je prikazan na sliki 5.17.

5.5 Krmiljenje robota

Sedaj, ko je robot umerjen, se lahko lotimo krmiljenja robota. Na začetku tako, da ga upravlja uporabnik preko svojega računalnika. Mreža tega programa je prikazana na sliki 5.18. Ob že znanima operacijama *CaptureNeuron* in *X11Neuron* imamo sedaj tudi operacijo *NXTSimpleNeuron*, ki preslikuje prejete krmilne podatke, v ukaze robotu. V primeru programa za umerjanje je to izvajala kar operacija *SetSteeringNeuron* sama, sedaj pa smo to



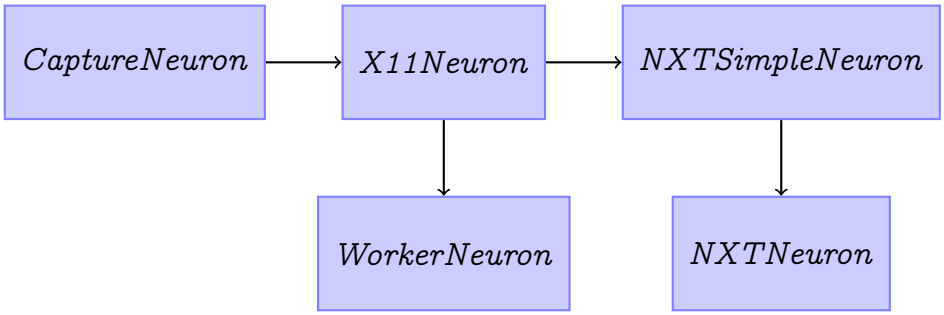
Slika 5.18: Mreža programa za krmiljenje robota.

```
1 main :: IO ()
2 main = do
3   prepareEnvironment
4   incubate $ do
5     nerveNXT ← (growNeuron :: NerveOnlyFor NXTSimpleNeuron) defaultOptions
6     nerveCapture ← (growNeuron :: NerveOnlyFrom CaptureNeuron) defaultOptions
7     nerveX11 ← (growNeuron :: NerveBoth X11Neuron) $ λo. o {targetEnabled = False}
8     nerveCapture `attachTo` [TranslatableFor nerveX11]
9     nerveX11 `attachTo` [TranslatableFor nerveNXT]
```

Slika 5.19: Koda programa za krmiljenje robota.

abstrahirali v *NXTSimpleNeuron*. Celoten program deluje tako, da operacija *X11Neuron* sprejema podatke o položaju robota, le-te prikazuje uporabniku, uporabnik se na podlagi njih odloča kako bo vodil robota, kar preko tipkovnice sporoča operaciji *NXTSimpleNeuron*.

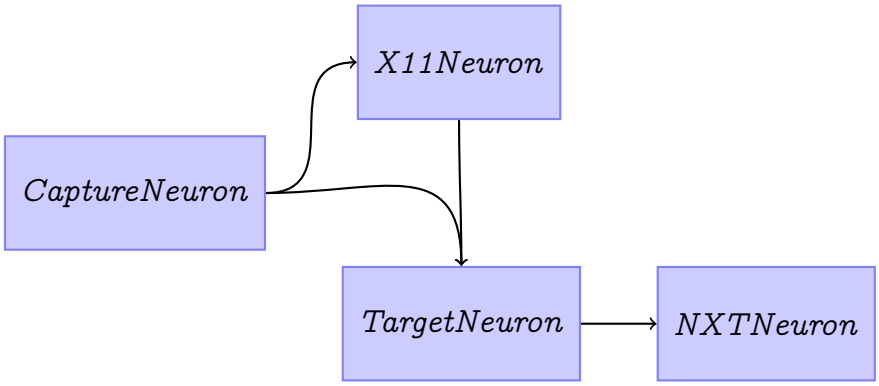
Program, prikazan v celoti na sliki 5.19, je kratek in enostaven, saj modularno uporablja operacije, v katerih je zakrita vsa morebitna kompleksnost delovanja. Tako se lahko tudi to delovanje prilagaja po potrebi, izboljšuje, medtem ko lahko ostane sam program nespremenjen. Naprimer, operacijo *NXTSimpleNeuron* bi lahko tako razširli s funkcionalnostjo umerjanja operacije *SetSteeringNeuron*, tako da bi robot med samo vožnjo primerjal realni kot svojega gibanja s pričakovanim in ga popravljal. V resnici je operacija *NXTSimpleNeuron* implementirana nad bolj splošno operacijo *NXTNeuron*. *NXTSimpleNeuron* omogoča le vnaprej definirane premike naprej, nazaj, levo in desno, medtem ko *NXTNeuron* omogoča poljubne



Slika 5.20: Mreža programa za krmiljenje robota s prikazanimi skritimi operacijami.

premike različnih velikosti, ampak še vedno z abstrahirano konkretno zasnovo robota, ki je v uporabi. Če smo natančni, je v resnici mreža programa bolj podobna tej na sliki 5.20. Pa še tam niso prikazane nekatere operacije, ki jih ogrodje samo interno uporablja. Operacija tako lahko zgradi svojo interno podmrežo (lahko tudi le ene operacije), ki jo uporablja na enak način, kot glaven program osnovno mrežo. *X11Neuron* naprimer uporablja *WorkerNeuron* za risanje podatkov na zaslon, tako da se lahko sama hitro odziva na pritiske tipk na tipkovnici. Operacija *X11Neuron* podatke za izris le pripravi, dokončno izračunajo in izrišejo pa se v operaciji *WorkerNeuron*.

Modularnost je seveda lastnost večine programskih jezikov in knjižnic. V imperativnih programskih jezikih se ponavadi definira programski vmesnik v obliki (imperativnih) funkcij, ki sprejemajo nek nabor argumentov, in ki se jih ponuja v obliki takšnih ali drugačnih hierarhij (naprimer razredov). V našem primeru ogrodja *Etage* je programski vmesnik definiran z operacijami, tipi podatkov, nad katerimi te operacije znajo delovati in obstoječimi definicijami prevajanja med različnimi tipi podatkov. Pomembno je, da le te definicije prevajanja med tipi podatkov definiramo naknadno, neodvisno od definiciji samih operacij oziroma samih tipov podatkov. To nam omogoča, da povezujemo operacije med seboj tudi s tistimi, ki si jih ob sami definiciji



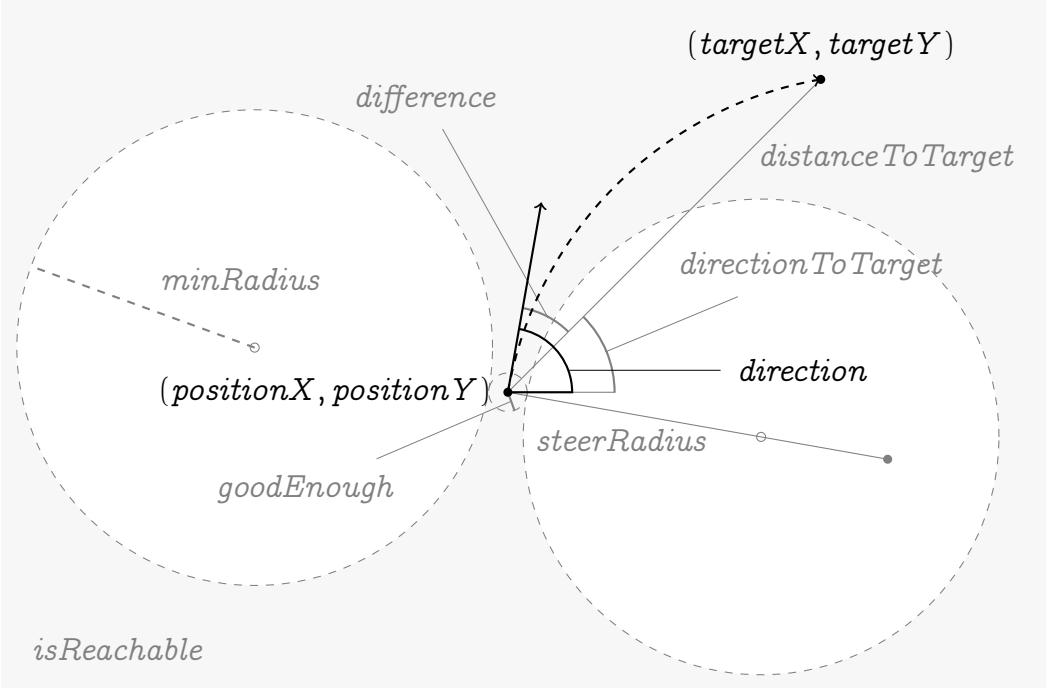
Slika 5.21: Mreža programa za vožnjo robota do tarče.

operacije še niti zamislili nismo.

5.6 Vožnja do tarče

Še bolj zanimivo je, če robota krmili kar program sam. Primer takšnega programa je lahko avtomatizirana vožnja do tarče, torej s strani uporabnika izbrane lokacije na poligonu. Program se lahko sestavi kot je prikazano na sliki 5.21. Operacije *CaptureNeuron* in *X11Neuron* že poznamo. Prav tako, kot že omenjeno, je bila operacija *NXTSimpleNeuron* le poenostavljena operacija *NXTNeuron*. Računanje potrebnih premikov robota pa se izvaja v operaciji *TargetNeuron*. Tako gredo podatki o položaju robota v operacijo *X11Neuron*, kjer se prikazujejo. Iz operacije *X11Neuron* pa gre v *TargetNeuron* podatek o lokaciji tarče s strani uporabnika. Operacija *TargetNeuron* prav tako potrebuje podatke o položaju robota, na podlagi katerih pošilja podatke o premikih operaciji *NXTNeuron*.

Celotna koda operacije *TargetNeuron* in programa je na voljo v dodatku B. Iz podatkov o položaju robota in lokaciji tarče redno računa potrebne premike robota, da robot doseže tarčo. Oziroma da jo doseže zadosti

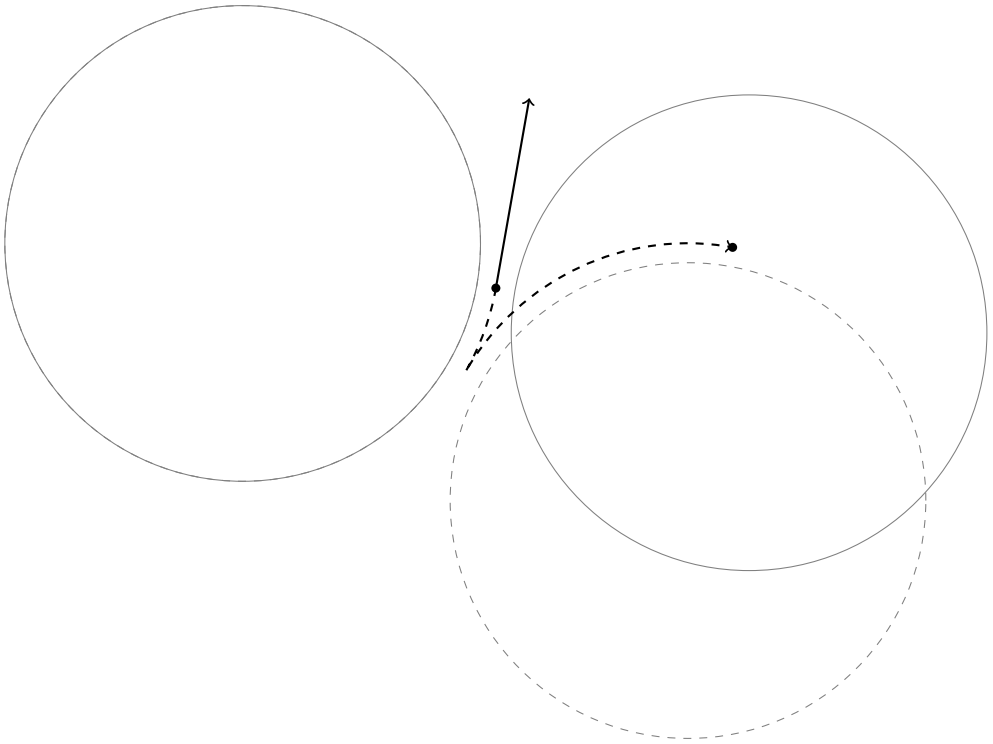


Slika 5.22: Izračun poti do tarče v funkciji *move*.

natančno.

Na sliki 5.22 je predstavljen uporabljen način usmerjanja k tarči, ki se računa v funkciji *move*. Za vhodne podatke ima podana položaj robota in lokacijo tarče. Iz tega izračuna polmer krožnice, ki bi glede na lokacijo in usmerjenost robota le-tega pripeljala neposredno do tarče. Iz radija se potem določi potreben kot koles in iz slednjega potreben premik motorja, ki nadzira kot koles. Le-ta se potem posreduje naprej po mreži programa do robota samega.

Vse skupaj se ponavlja tako hitro, kot prihajajo podatki o izpeljanih premikih robota nazaj do operacije, s čimer lahko operacija v realnem času popravlja morebitne odklone od izračunane poti do tarče, saj je odzivnost tako zaznave položaja kot posredovanja ukazov za premik robotu dovolj visoka. Tako se v splošnem robota pripelje v elegantnem in gladkem gibanju



Slika 5.23: Pot do tarče, če ta na začetku ni neposredno dostopna.

do tarče.

Žal obstajajo primeri, ko takšen način vožnje do tarče odpove. Glede na zasnovo robota in njegov način gibanja z zavijanjem koles ima namreč robot dve "slepi" področji, do katerih se ne more premakniti neposredno po krožnici, saj ne more koles obrniti za poljuben kot. Torej obstaja najmanjši polmer krožnice, ki ga lahko robot zvozi. Ta je na sliki 5.22 označen z *minRadius* in določa dve krožni površini, za kateri velja, da če je tarča znotraj njih, se robot do nje ne more premakniti neposredno po krožnici.

V tem primeru se mora najprej umakniti tako, da (čimprej) premakne tarčo v položaj glede na sebe tako, da jo lahko potem doseže neposredno. Primer takšne poti je prikazan na sliki 5.23. V splošnem je ta začetni premik precej majhen, saj dovoljujemo robotu premikanje tako naprej in nazaj

in tudi do tarče se lahko premakne ne glede na svojo usmeritev. Tako ima štiri možne smeri umikanja. S tem dodatnim načinom delovanja lahko robot doseže katerokoli točko v poligonu v elegantnem, mogoče ne najbolj pričakovanem, načinu (ljudje ponavadi obrnemo svoje vozilo najprej v smeri cilja in potem vozimo tja naravnost, ne pa v na teoriji osnovanem loku).

Vožnja robota do tarče tako predstavlja primer sistema s kompleksnim delovanjem, kjer pa je bilo potrebno razviti le to krmilno operacijo delovanja, medtem ko smo ostale le ponovno uporabili in na njih gledali kot na neodvisne že pripravljene module. In ne samo med razvojem, tudi med delovanjem programa ostale operacije delujejo neodvisno in vzporedno. Naprimer določanje novega položaja robota se dogaja neodvisno in hkrati kot računanje sedanjega premika, hkrati pa se pravkar izračunani premik pošilja robotu v izvedbo. O vseh delovanjih nam med razvojem krmilne operacije sploh ni bilo potrebno razmišljati, mogoče nam jih tudi niti ne bi bilo potrebno poznati. Na takšen način ogrodje omogoča modularnost že pri načrtovanju, razvoju in tudi samemu izvajanju programa.

Poglavje 6

Sklepne ugotovitve

V tem delu smo predstavili nedeterministično in leno ogrodje za podatkovno pretokovno programiranje in računanje. Namignili smo na nekaj primerov uporabe takšnega ogrodja in predstavili dva večja primera, kjer eden izkorišča podobnost mreže operacij in povezav z matematično strukturo grafov, drugi pa prikazuje uporabnost in modularnost v problemskih domenah, ki temeljijo na dogodkih in tokovih podatkov, kar je naprimer krmiljenje sistema, v našem primeru robota, v realnem času. Za kar se je naša delovna hipoteza o primernosti takšnega pristopa izkazala za upravičeno.

Razvili smo kar nekaj sistemov, od ogrodja samega do podpornih programov za njegovo delovanje in njegovo testiranje. V okviru ogrodja smo podatkovno pretokovni model razširili v dogodkovno pretokovnega. Za potrebe krmiljenja robota smo razvili zajem položaja robota preko spletne kamere in kalibracijo zajema samega. In za iskanje najcenejših poti v grafu dodatek k ogrodju za delo z grafi. Pri tem smo prispevali popravke in razširitve tudi k raznim odprtokodnim sistemom, ki smo jih v okviru tega dela uporabili. Kot tudi prosto objavili izvorno kodo ogrodja na Internetu. Vse to nam je omogočilo, da so rezultati tega dela vidni in dostopni tudi drugje, na voljo

drugim, da jih raziščejo, sodelujejo in/ali na njem delajo naprej.

Dotaknili smo se tudi zmogljivostne analize delovanja ogrodja in programov na njegovi osnovi. Nekako po pričakovanjih se je izkazal za počasnejšega od direktne ukazno pretokovne implementacije, ampak tu obstaja še veliko možnosti za nadaljnje izboljšave, naprimer uporabo raznih drugih implementacij pošiljanja podatkov po mreži. Hkrati se je že sedaj ogrodje izkazalo primerno za sisteme z več CPE, saj omogoča enostavno in avtomatično vzporedno izvajanje programa.

Poglaviten poudarek našega dela je, da se lahko s pomočjo podatkovno pretokovnega modela enostavno modelirajo in zatem tudi implementirajo kompleksni sistemi, brez potrebe po izjemnih primerih ali prehodih med različnimi modalnostmi in (pod)sistemi. V tem modelu ni razlike med uporabnikom in programom, med "zunaj" in "znotraj" računalnika, ker je vse le ena mreža operacij in povezav med njimi, po katerih se pretakajo podatki. Operacije so definirane nedeterministično, s tem druge operacije to pričakujejo oziroma podpirajo, zato na takšen način ni problem modelirati tudi naravnih procesov, naključnih spremenljivk, pa tudi uporabnika samega. Tudi računalniška omrežja lahko predstavimo na enak način. Primer mreže takšnega razširjenega pogleda na podatkovno pretokovni model je prikazan na sliki 6.1.

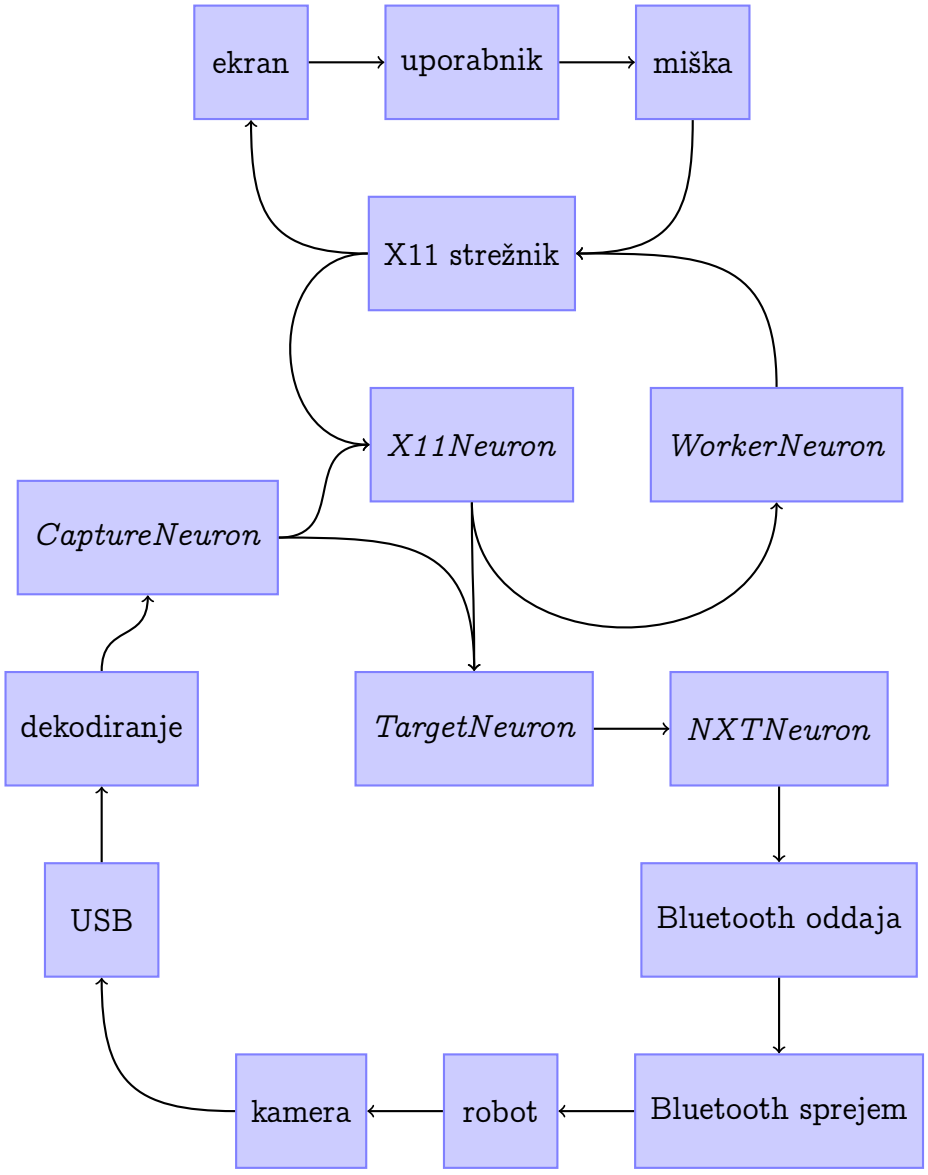
Lenost ogrodja ni bila pretirano izpostavljena. Zanimiva lastnost je namreč, da medtem ko se informacija o obstoju podatka prenaša po povezavah, v realnem času, se sama vsebina podatka izračuna leno, le ko in če je sploh potrebna. To preprečuje nepotrebno računanje vrednosti, ki jih program mogoče niti več ne potrebuje, ker so naprimer zastarele. Zanimivo bi bilo globlje raziskati možnosti izrabe lenosti za dodatno zmogljivostno izboljšavo ogrodja oziroma sploh natančno analizirati interakcijo lenosti s samim delo-

vanjem ogrodja.

Ob tem se je potrebno zavedati, da ogrodje samo predvsem ponuja pristop k rešitvi in ne rešitve same. Tako je trenutno za veliko problemskih področij potrebno razviti operacije samostojno, saj samih knjižnic na osnovi orodja, ki bi vse to že ponujale, še ni. Z večjim številom knjižnic z raznih področij pa se bo šele lahko videla prednost v modularnosti in enostavnosti ponovne uporabe, ki jo ogrodje omogoča. Na srečo pa je pogosto mogoče zaobjeti obstoječo ustrezno imperativno knjižnico v operacijo ogrodja, ter jo vsaj na takšen način ponuditi v uporabo programom na osnovi našega ogrodja.

Prav tako v predstavljenih primerih niso najbolje vidne možnosti večnivojske gradnje mrež. Naprimer, v primeru krmiljenja robota si lahko zamišljamo osnoven nivo delovanja robota, nad katerim se potem ustvari nova mreža, ki doda višjenivojsko procesiranje senzoričnih podatkov, druga, ki višjenivojsko procesira robotovo motoriko, ter nad obema še dodatna, ki se uči in ga usmerja na podlagi pridobljenega znanja. Slednja bi lahko bila tudi povezana z že obstoječimi podatkovno pretokovnimi sistemi za strojno učenje in obdelavo podatkov, kot je naprimer Orange [7]. Tako lahko nivoji naravno sovpadajo z večnivojskostjo problemske domene. Prav tako bi nivoji lahko omogočali tudi deljenje programa na čisto delovanje (naprimer interno procesiranje) ter interakcijo z okoljem, ter tudi pomagali, da se kakšno nezaželjeno predznanje ne posreduje nivoju za učenje.

Pristop v okviru ogrodja *Etage* k podatkovno pretokovnemu računanju se je pokazal za zelo delujočega, uporabnega in zanimivega. Odpira nove možnosti raziskovanja in omogoča popolnoma splošno in enotno abstrakcijo še tako kompleksnih sistemov. Zato predlagamo, da se še naprej razvija, raziskuje ter ugotavlja njegove prednosti in slabosti. Predvsem da se preizkusi še na drugih problemskih domenah, naprimer porazdeljenemu računanju



Slika 6.1: Razširjen in še vedno nepopoln prikaz mreže sistema za vožnjo robota do tarče.

oziroma delovanju, procesiranju signalov (naprimer zvočnih), za vmesnike človek-računalnik in nasploh vse sisteme, ki so povezani s fizičnim svetom . . .

Literatura

[1] Heinrich Apfelmus. The Operational Monad Tutorial. *The Monad.Reader*, Issue 15:37–55, Januar 2010. Dostopno na `http://themonadreader.wordpress.com/2010/01/26/issue-15/`. 44

[2] Joe Armstrong. A history of Erlang. Objavljeno v *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, strani 6-1–6-26, New York, NY, USA, 2007. ACM. 16

[3] James Cheney in Ralf Hinze. First-Class Phantom Types. Technical Report 1901, Cornell University, Julij 2003. Dostopno na `http://ecommons.library.cornell.edu/bitstream/1813/5614/1/TR2003-1901.pdf`. 36

[4] Juliusz Chroboczek. The Babel routing protocol. RFC 6126 (Experimental), April 2011. Dostopno na `http://www.ietf.org/rfc/rfc6126.txt`. 55

[5] Antony Alexander Courtney. *Modeling user interfaces in a functional language*. Doktorska dizertacija, Yale University, New Haven, CT, USA, 2004. 17

[6] Antony Alexander Courtney in Conal Elliott. Genuinely Functional User Interfaces. Objavljeno v *Proceedings of the 2001 Haskell Workshop*, September 2001. Dostopno na `http://conal.net/papers/genuinely-functional-guis.pdf`. 17

[7] Janez Demšar, Gregor Leban, in Blaž Zupan. Orange: From Experimental Machine Learning to Interactive Data Mining, 2004. White Paper. Dostopno na `http://orange.biolab.si/wp/orange.pdf`. 92

[8] Anthony Discolo, Tim Harris, Simon Marlow, Simon L. Peyton Jones, in Satnam Singh. Lock Free Data Structures using STMs in Haskell. Objavljeno v *Proceedings of the Eighth International Symposium on Functional and Logic Programming*, FLOPS '06, strani 65–80. Springer, 2006. Dostopno na `http://research.microsoft.com/pubs/67420/2006-flops.pdf`. 14, 30, 61

[9] Conal Elliott. Push-pull functional reactive programming. Objavljeno v *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, strani 25–36, New York, NY, USA, 2009. ACM. Dostopno na `http://conal.net/papers/push-pull-frp/push-pull-frp.pdf`. 13, 16

[10] Conal Elliott in Paul Hudak. Functional reactive animation. Objavljeno v *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, strani 263–273, New York, NY, USA, 1997. ACM. Dostopno na `http://conal.net/papers/icfp97/icfp97.pdf`. 17

[11] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11:467–492, September 2001. Dostopno na `http://web.engr.oregonstate.edu/~erwig/papers/InductiveGraphs_JFP01.pdf`. 54, 59

[12] Tim Harris, Simon Marlow, Simon L. Peyton Jones, in Maurice Herlihy. Composable memory transactions. Objavljeno v *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, strani 48–60, New York, NY, USA, 2005. ACM. Dostopno na `http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/stm.pdf`. 14, 30, 61

[13] Carl Hewitt, Peter Bishop, in Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. Objavljeno v *Proceedings of the 3rd international joint conference on Artificial intelligence*, strani 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. Dostopno na `http://ijcai.org/Past%20Proceedings/IJCAI-73/PDF/027B.pdf`. 15

[14] Paul Hudak, Antony Alexander Courtney, Henrik Nilsson, in John Peterson. Arrows, Robots, and Functional Reactive Programming. Objavljeno v *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, strani 159–187. Springer, 2002. Dostopno na `http://haskell.cs.yale.edu/yampa/AFPLectureNotes.pdf`. 16, 17

[15] Paul Hudak, John Hughes, Simon L. Peyton Jones, in Philip Wadler. A history of Haskell: being lazy with class. Objavljeno v *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, strani 12-1–12-55, New York, NY, USA, 2007. ACM. Dostopno na `http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf`. 7

[16] John Hughes. *Why functional programming matters*, strani 17–42. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. Dostopno na `http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf`. 10, 29

[17] Sheng Liang, Paul Hudak, in Mark Jones. Monad transformers and modular interpreters. Objavljeno v *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, strani 333–343, New York, NY, USA, 1995. ACM. Dostopno na `http://web.cecs.pdx.edu/~mpj/pubs/modular-interpreters.ps`. 9

[18] Andres Löh in Ralf Hinze. Open data types and open functions. Objavljeno v *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, strani 133–144, New York, NY, USA, 2006. ACM. Dostopno na `http://people.cs.uu.nl/andres/OpenDatatypes.pdf`. 10

[19] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, in Philip W. Trinder. Seq no more: better strategies for parallel Haskell. Objavljeno v *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, strani 91–102, New York, NY, USA, 2010. ACM. Dostopno na `http://www.macs.hw.ac.uk/~dsg/gph/papers/pdf/new-strategies.pdf`. 13, 59

[20] Izzet Pembeci. *Functional reactive programming as a framework for hybrid systems and robot programming.* Doktorska dizertacija, The Johns Hopkins University, Baltimore, MD, USA, 2004. 17

[21] John Peterson, Paul Hudak, in Conal Elliott. Lambda in Motion: Controlling Robots with Haskell. Objavljeno v *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, strani 91–105, London, UK, 1998. Springer. Dostopno na `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.8118&rep=rep1&type=pdf`. 17

[22] Simon L. Peyton Jones et al. The Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(1):0–255, Januar 2003. Dostopno na `http://www.haskell.org/definition/`. 7

[23] Simon L. Peyton Jones, Andrew Gordon, in Sigbjorn Finne. Concurrent Haskell. Objavljeno v *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, strani 295–308, New York, NY, USA, 1996. ACM. Dostopno na `http://research.microsoft.com/en-us/um/people/simonpj/papers/concurrent-haskell.ps.gz`. 14

[24] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, in Phil Wadler. The Glasgow Haskell compiler: a technical overview. Objavljeno v *Proceedings of Joint Framework for Information Technology Technical Conference*, strani 249–257, London, UK, 1993. DTI/SERC. Dostopno na `http://www.research.microsoft.com/Users/simonpj/Papers/grasp-jfit.ps.Z`. 15

[25] Simon L. Peyton Jones in Philip Wadler. Imperative functional programming. Objavljeno v *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, strani 71–84, New York, NY, USA, 1993. ACM. Dostopno na `http://homepages.inf.ed.ac.uk/wadler/papers/imperative/imperative.ps`. 9

[26] Tom Schrijvers, Simon L. Peyton Jones, Manuel Chakravarty, in Martin Sulzmann. Type checking with open type functions. Objavljeno v *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, strani 51–62, New York, NY, USA, 2008. ACM. Dostopno na `http://www.cse.unsw.edu.au/~chak/papers/tc-tfs.pdf`. 9

[27] Tom Schrijvers, Simon L. Peyton Jones, Martin Sulzmann, in Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. Objavljeno v *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, strani 341–352, New York, NY, USA, 2009. ACM. Dostopno na `http://research.microsoft.com/en-us/um/people/simonpj/papers/gadt/implication_constraints.pdf`. 36

[28] Martin Sulzmann, Edmund S. L. Lam, in Peter Van Weert. Actors with multi-headed message receive patterns. Objavljeno v *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION '08, strani 315–330. Springer, 2008. Dostopno na `http://www.comp.nus.edu.sg/~lamsoonl/papers/actor-multimessage.ps`. 16

[29] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, in Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Januar 1998. Dostopno na `http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/strategies.ps`. 13, 59

[30] Emanuele Trucco in Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998. 71

[31] Jurij Šilc, Borut Robič, in Theo Ungerer. *Processor architecture: from dataflow to superscalar and beyond*. Springer, 1999. 3, 13

[32] Philip Wadler. The expression problem, 1998. Prispevek na java-genericity poštnem seznamu. Dostopno na `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`. 10

[33] Philip Wadler in Stephen Blott. How to make ad-hoc polymorphism less ad hoc. Objavljeno v *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, strani 60–76, New York, NY, USA, 1989. ACM. Dostopno na `http://homepages.inf.ed.ac.uk/wadler/papers/class/class.ps`. 8

Dodatek A

Program za računanje najcenejše poti v grafu


```
1  module Data.Graph.Etage (
2     shortestPaths,
3     sendTopologyChange,
4     GraphImpulse (..)
5  ) where

6  import Control.Exception
7  import Control.Monad.State
8  import Data.Data
9  import Data.Graph.Inductive hiding (inn, inn', out, out', node', nodes, run)
10 import qualified Data.Map as M
11 import Data.Map hiding (filter, map, empty, null, lookup)
12 import Data.Tuple
13 import System.IO

14 import Control.Etage
```

```
{- Funkcija, ki preslika graf v mrežo povezav in operacij, kjer vsako vozlišče grafa dobi svojo ustrezno operacijo, povezave grafa pa se preslikajo v povezave mreže. Rezultat je preslikava med vozlišči grafa in ustreznimi povezavami na ustrezajoče operacije mreže, kjer se lahko kasneje povežemo na željeno povezavo in spremljamo računanje najcenejših poti glede na s tem izbrano operacijo mreže – vozlišče grafa. -}
shortestPaths :: (DynGraph gr, Show a, Data a, Data b, Real b, Bounded b) ⇒
  gr a b → Incubation (M.Map Node (Nerve (GraphImpulse a b) AxonConductive (GraphImpulse a b) AxonConductive))
shortestPaths = ufoldM' growGraph M.empty

{- Pomožna funkcija, ki deluje ob pregibanju nad grafom, za preslikavo grafa v mrežo povezav in operacij. Ob pregibanju se deluje posamično na vsako vozlišče grafa in na njega povezane povezave. Te ta pomožna funkcija ustrezno preslika v operacijo in povezave, ki jih tudi ustrezno poveže, v obe smeri. -}
growGraph :: ∀a b.(Show a, Data a, Data b, Real b, Bounded b) ⇒
  Context a b → M.Map Node (Nerve (GraphImpulse a b) AxonConductive (GraphImpulse a b) AxonConductive) →
    Incubation (M.Map Node (Nerve (GraphImpulse a b) AxonConductive (GraphImpulse a b) AxonConductive))
growGraph (inn, node, label, out) nodes = do
  liftIO $ do
    assertIO $ node `notMember` nodes -- trenutno vozlišče še nismo obdelali
    assertIO $ all ((`member` nodes) ∘ snd) inn' -- že smo obdelali vsa vozlišča vhodnih in izhodnih povezav
    assertIO $ all ((`member` nodes) ∘ snd) out' -- za to poskrbi pravilno pregibanje
   -- nova operacija za trenutno vozlišče
  nodeNerve ← (growNeuron :: NerveBoth (NodeNeuron a b)) (λo. o {lnode = (node, label)})
  mapM_ ((`attachTo` [TranslatableFor nodeNerve]) ∘ (nodes!) ∘ snd) out' -- povežemo izhodne povezave
  nodeNerve `attachTo` map (TranslatableFor ∘ (nodes!) ∘ snd) inn' -- povežemo vhodne povezave
  liftIO $ do
    time ← getCurrentImpulseTime
     -- obvestimo novo operacijo o njenih izhodnih povezavah
    unless (null out') $ sendForNeuron nodeNerve $ AddOutEdges time out'
     -- obvestimo druge operacije, ki imajo vhodno povezavo na to novo operacijo, o njihovi novi izhodni povezavi
    mapM_ (λ(l, n). sendForNeuron (nodes ! n) $ AddOutEdges time [(l, node)]) inn'
  return $ insert node nodeNerve nodes -- vstavimo povezavo nove operacije v preslikavo in jo vrnemo
```

```
39      where inn' = filter ((node ≢) ∘ snd) inn -- ignoriramo morebitne povezave nazaj na vozlišče samo
40            out' = filter ((node ≢) ∘ snd) out -- ignoriramo morebitne povezave nazaj na vozlišče samo

41 {- Obvesti vse operacije mreže o tem, da je prišlo do spremembe v topologiji grafa (sprememba cene povezave, nova/odstranjena
   povezava ali operacija). Trenutno se uporablja le takoj za prvo izgradnjo mreže, saj programski vmesnik za spreminjanje topologije
   še ni na voljo uporabniku. -}

42 sendTopologyChange ::
43   M.Map Node (Nerve (GraphImpulse a b) AxonConductive (GraphImpulse a b) AxonConductive) → Incubation ()
44 sendTopologyChange nodes = liftIO $ do
45   time ← getCurrentImpulseTime
46   forM_ (elems nodes) $ λn.
47     sendForNeuron n $ TopologyChange time

48 {- Glavna funkcija, ki izvaja osnovno delovanje vozlišča/operacije. -}
49 run :: (Data b, Real b, Bounded b) ⇒
50   Nerve (GraphImpulse a b) fromConductivity (GraphImpulse a b) forConductivity → NodeNeuron a b → NodeIO a b ()
51 run nerve (NodeNeuron node label) = forever $ do  -- v neskončnost ponavljamo
52   impulse ← liftIO $ getForNeuron nerve -- preberemo naslednji podatek namenjen temu vozlišču/operaciji
53   case impulse of  -- če je podatek ...
54     -- ... informacija o spremembi topologije
55     TopologyChange {impulseTimestamp} → do
56       lastTimestamp ← gets lastTopologyChangeTimestamp -- pridobimo iz stanja čas zadnjega takšnega podatka
57       when (impulseTimestamp > lastTimestamp) $ do  -- le če je nov podatek novejši od zadnjega, s tem preprečimo cikle
58         modify (λs. s {lastTopologyChangeTimestamp = impulseTimestamp}) -- shranimo čas novega podatka v stanje
59         paths ← gets currentPaths -- pridobimo iz stanja trenutno znane poti
60         liftIO $ do
61           sendFromNeuron nerve impulse -- pošljemo ta podatek naprej
62           t ← liftIO getCurrentImpulseTime
63           -- pošljemo vse trenutno znane poti naprej (morebitnim novim vozliščem/operacijam)
64           forM_ (toList paths) $ λ(n, (l, p)).
65             sendFromNeuron nerve
```

```
66                TopologyUpdate {impulseTimestamp = t, originator = (node, label), destination = (n, l), path = p}
67      -- ... informacija o (novi) najcenejši poti
68      TopologyUpdate {impulseTimestamp, originator = (o, _), destination = (d, l), path = (LP path, cost)} → do
69        liftIO $ do
70          -- shranjena cena celotne poti se mora ujemati s potjo samo; pri tem upoštevamo morebitne zaokrožitvene napake
71          assertIO $ abs (cost − (sum ∘ map snd $ path)) · 100000 < 1
72          assertIO $ (fst ∘ last $ path) ≡ d -- zadnje vozlišče poti mora biti enako končnemu vozlišču podanem v podatku
73        out ← gets outedges -- pridobimo v stanju shranjene povezave na to vozlišče in njihove cene
74        case M.lookup o out of
75          Nothing   → -- podatki o povezavi na to vozlišče bi morali že biti na voljo
76            liftIO $ hPutStrLn stderr "Warning: TopologyUpdate message arrived before AddOutEdges message."
77          Just ocost → do -- sicer pogledamo, če ta (nova) pot omogoči tudi nam izboljšavo do sedaj znane poti
78            paths ← gets currentPaths -- pridobimo iz stanja trenutno znane poti
79            let -- najdemo temu vozlišču/operaciji trenutno znano ceno poti, oziroma zanjo izberemo zgornjo mejo cene
80                (_, (_, c)) = findWithDefault (⊥, (⊥, maxBound)) d paths
81                cost'       = cost + ocost -- izračunamo ceno poti, če bi uporabili to (novo) pot
82            when (cost' < c) $ do -- če je cena preko te (nove) poti nižja
83              let path'  = LP $ (node, ocost) : path
84                  paths' = insert d (l, (path', cost')) paths
85              modify (λs. s {currentPaths = paths'}) -- shranimo znane poti s to novo izboljšano/cenejšo potjo v stanje
86               -- pošljemo to novo izboljšano pot naprej
87              liftIO $ sendFromNeuron nerve
88                TopologyUpdate {impulseTimestamp, originator = (node, label), destination = (d, l), path = (path', cost')}
89      -- ... informacija o novih povezavah in njihovih cenah
90      AddOutEdges {newOutEdges} → do
91        out ← gets outedges -- pridobimo v stanju shranjene povezave na to vozlišče in njihove cene
92        let out' = foldl (λi (l, n). insert n l i) out newOutEdges -- dodamo nove povezave
93        modify (λs. s {outedges = out'}) -- shranimo posodobljene povezave in njihove cene v stanje
```

```
94 {- ========== Od tu dalje sledijo razne pomožne definicije. ========== -}

95 type SPath b = (LPath b, b) -- urejen par poti in skupne cene te poti
96 type SPaths a b = M.Map Node (a, SPath b) -- preslikava med končnim vozliščem in potjo

97 {- Podatkovni tip stanja operacije/vozlišča. -}
98 data NodeState a b = NodeState {
99       lastTopologyChangeTimestamp :: ImpulseTime,
100      currentPaths :: SPaths a b, -- preslikava med trenutno znanimi končnimi vozlišči in izračunanimi najboljšimi potmi do njih
101      outedges :: M.Map Node b -- preslikava med izhodnimi povezavami in njihovimi cenami
102    }

103 type NodeIO a b = StateT (NodeState a b) IO

104 {- Podatkovni tip operacije. Vsebuje vozlišče in njegovo oznako. -}
105 data NodeNeuron a b = NodeNeuron Node a deriving (Typeable, Data)

106 deriving instance Typeable1 LPath
107 deriving instance Data a => Data (LPath a)

108 {- Podatkovni tip podatka, ki ga uporabljamo za posredovanje informacij o iskanju najcenejših poti. -}
109 data GraphImpulse a b = TopologyUpdate {
110      impulseTimestamp :: ImpulseTime,
111      originator :: LNode a, -- vozlišče/operacija, ki pošilja podatek
112      destination :: LNode a, -- končno vozlišče za katerega se pošilja nova najcenejša znana pot
113      path :: SPath b -- trenutno najcenejša znana pot pošiljajočega vozlišča do končnega vozlišča
114    } | TopologyChange {
115      impulseTimestamp :: ImpulseTime
116    } | AddOutEdges {
117      impulseTimestamp :: ImpulseTime,
118      newOutEdges :: Adj b -- nove povezave in njihove cene
119    } deriving (Eq, Ord, Show, Typeable, Data)
```

```
instance (Show a, Typeable a, Show b, Typeable b, Real b, Bounded b) ⇒ Impulse (GraphImpulse a b) where
  impulseTime TopologyUpdate {impulseTimestamp} = impulseTimestamp
  impulseTime TopologyChange {impulseTimestamp} = impulseTimestamp
  impulseTime AddOutEdges    {impulseTimestamp} = impulseTimestamp
  impulseValue TopologyUpdate {originator = (o, _), path} = toRational o : (value ∘ fst $ path)
    where value (LP p) = concatMap (λ(n, l). [toRational n, toRational l]) p
  impulseValue TopologyChange {}                          = []
  impulseValue AddOutEdges    {newOutEdges}               = concatMap value newOutEdges
    where value (l, n)    = [toRational l, toRational n]

instance (Show a, Data a, Show b, Data b, Real b, Bounded b) ⇒ Neuron (NodeNeuron a b) where
  type NeuronFromImpulse (NodeNeuron a b) = GraphImpulse a b
  type NeuronForImpulse (NodeNeuron a b)  = GraphImpulse a b
  data NeuronOptions (NodeNeuron a b) = NodeOptions {
    lnode :: LNode a
    } deriving (Eq, Ord, Read, Show)

  mkDefaultOptions = return NodeOptions {
    lnode = ⊥
    }

  grow NodeOptions {lnode = (node, label)} = return $ NodeNeuron node label

  live nerve neuron@(NodeNeuron node label) =
    -- začetno stanje vsebuje le edino trenutno znano pot – do tega vozlišča/operacije same, cene 0
    evalStateT (run nerve neuron) (NodeState 0 (singleton node (label, (LP [(node, 0)], 0))) M.empty)

{- Funkcija pregibanja nad grafom, definira za uporabo nad vrednostmi v monadi. -}
ufoldM' :: (Graph gr, Monad m) ⇒ (Context a b → c → m c) → c → gr a b → m c
ufoldM' f u g | isEmpty g = return u
              | otherwise = ufoldM' f u g' >>= λu'. f c u'
                  where (c, g') = matchAny g
```

Dodatek B

Program za vožnjo robota do tarče


```
 1 module Target (
 2     TargetNeuron,
 3     TargetFromImpulse (..),
 4     TargetForImpulse (..),
 5     TargetOptions,
 6     NeuronOptions,
 7     main
 8 ) where

 9 import Control.Applicative
10 import Control.Monad.State
11 import Data.Data
12 import Data.Fixed
13 import Data.Maybe

14 import Control.Etage
```

```
15 import Capture
16 import NXT
17 import X11

18 {- Kako blizu tarče nam je dovolj blizu? -}
19 goodEnough :: Double
20 goodEnough = 2 -- cm

21 {- Za koliko se mora robot premakniti, da ponovno ocenimo njegov položaj napram tarči? -}
22 minDistanceChange :: Double
23 minDistanceChange = 0.1 -- cm

24 {- Funkcija izračuna radij krožnice iz znane tangente nanjo (točka na tangenti in smer, v našem primeru lokacija robota in njegova
   usmerjenost) in točke na krožnici (v našem primeru lokacija tarče). Podpira tudi odmik, to je pravokotno razdaljo od točke na
   tangenti. Ta radij uporabljamo za izračun velikosti zavoja koles robota. Funkcija je bila pridobljena analitično. -}

25 tangentRadius :: (Double, Double) → Double → (Double, Double) → Double → Double
26 tangentRadius (positionX, positionY) direction (targetX, targetY) offset = radius
27   where radius = ((positionX − targetX)^2 + (positionY − targetY)^2 + offset^2 − 2 · offset · s) / (2 · (s − offset))
28         s      = max (abs $ cos direction · (positionY − targetY) − sin direction · (positionX − targetX)) offset

29 {- Funkcija, ki izračuna morebitni potrebni premik robota glede na njegovo trenutni položaj in lokacijo tarče. Ker je funkcija
   čista, se lahko opazi deklerativna narava njene definicije. Ob tem se je potrebno tudi spomniti, da je Haskell len programki jezik,
   kar pomeni, da se izračunajo le tiste vrednosti, ki jih potrebuje za izračun končnega rezultata funkcije. -}

30 move :: TargetForImpulse → TargetForImpulse → Maybe (Bool, Int, Int)
31 move TargetCapture {position = Robot {positionX, positionY, direction}} TargetPosition {targetX, targetY}
32     -- če smo že dovolj blizu tarče, se ustavimo
33     | distanceToTarget < goodEnough = Nothing
34     -- če je tarča neposredno dosegljiva iz trenutnega položaja, se premaknemo proti njej
35     | isReachable                   = Just (forwardsOrBackwards, fromIntegral steerTo, speed)
36     -- sicer se umaknemo tako, da postane neposredno dosegljiva
37     | otherwise                     = Just (reachForwardsOrBackwards, fromIntegral reachSteerTo, maxSpeed)
38   where distanceToTarget = sqrt $ (positionX − targetX)^2 + (positionY − targetY)^2 -- razdalja do tarče
```

```
39  -- smer do tarče, glede na koordinatni sistem poligona
40  directionToTarget = atan2 (targetY − positionY) (targetX − positionX)
41  -- smer do tarče, glede na trenutno usmerjenost robota, torej kot do tarče glede na robota
42  difference = angleRange $ directionToTarget − direction
43    where angleRange a = ((a + π) `mod` (2 · π)) − π -- kot si želimo v [−π, π) intervalu
44  difference' = difference / (π / 2) -- uporaben pa je tudi v [−2, 2) intervalu
45  -- glede na kot se odločimo, ali se mora robot premikati naprej ali nazaj, da najhitreje doseže tarčo
46  forwardsOrBackwards = (π / 2) ⩾ difference ∧ difference ⩾ (−π / 2)
47  -- radij krožnice (poti) do tarče
48  steerRadius = tangentRadius (positionX, positionY) direction (targetX, targetY) 0
49  -- ali krožnico (poti) do tarče lahko zvozimo z uporabljeno konstrukcijo robota,
50  -- kjer je minRadius polmer najbolj ostrega zavoja, ki ga robot zmore
51  isReachable = tangentRadius (positionX, positionY) direction (targetX, targetY) goodEnough >
52                    minRadius − goodEnough
53  -- iz radija izračunamo kot koles glede na robota, wheelbase je medosna razdalja
54  steerAngle = asin $ (wheelbase / 2) / steerRadius
55  -- izračunamo utež kota, saj si želimo, da se pri velikih kotih robot obrne bolj, kot bi se najmanj moral,
56  -- da čimprej pride v vožnjo v smeri tarče
57  weightDifference = min (2 − abs difference') (abs difference')
58  steerAngle' = steerAngle · exp (weightDifference^4) -- obtežen kot koles
59  -- izračunamo premik motorja iz željenega kota koles
60  steerTo = (round ∘ signum $ difference) · steerFromAngle steerAngle'
61  -- hitrost izračunamo tako, da bližje kot je robot tarči, bolj počasi (previdno) se vozi, da ne zgreši tarče
62  speed = round $ (distanceToTarget / goodEnough − 1) · (fromIntegral maxSpeed / (minRadius / goodEnough)) +
63          (fromIntegral maxSpeed / 4)
64  -- v primeru, da se robot mora umikati, da pride v položaj, kjer je tarča neposredno dosegljiva,
65  -- se glede na kot do tarče (oziroma kvadrant tega kota) poskuša čimhitreje premakniti v takšen položaj
66  (reachSteerTo, reachForwardsOrBackwards) = case floor (difference' `mod` 4) :: Int of
67                                                0 → ( − maxSteer, False)
68                                                1 → ( − maxSteer, True)
```

```
69                                                    2 → (    maxSteer, True)
70                                                    3 → (    maxSteer, False)
71 {- Glavna funkcija, ki izvaja osnovno delovanje operacije usmerjanja robota do tarče. -}
72 run :: Nerve TargetFromImpulse fromConductivity TargetForImpulse forConductivity → Target ()
73 run nerve = forever $ do  -- v neskončnost ponavljamo
74   TargetState {target, capture} ← get -- pridobimo trenutno stanje
75    -- preberemo le najnovejše podatke, ki so na voljo, glede na posamezne konstruktorje (TargetCapture in TargetPosition)
76    -- in posodobimo trenutno znane podatke o tarči in položaju robota
77   (target', capture') ← liftIO $ squash (target, capture) ⟨$⟩ getNewestForNeuron nerve
78   when (target ≢ target' ∨ distanceChanged capture capture') $ do  -- če se je spremenila tarča ali se je robot premaknil
79     when (isJust capture' ∧ isJust target') $ liftIO $ do  -- če imamo vse potrebne podatke (položaj robota in tarčo)
80       impulseFromTimestamp ← getCurrentImpulseTime
81        -- izračunamo morebitni potrebni premik in ga pošljemo naprej
82       sendFromNeuron nerve $ case move (fromJust capture') (fromJust target') of
83         Just (forwardsOrBackwards, steerTo, speed) →
84           Move {impulseFromTimestamp, forwardsOrBackwards, steerTo, speed}
85         Nothing                                    →
86           Stop  {impulseFromTimestamp}
87     put TargetState {target = target', capture = capture'} -- shranimo posodobljeno stanje
88 where  -- izračun razdalje med dvema položajema robota in primerjava z mejno vrednostjo
89       distanceChanged (Just TargetCapture {position = Robot {positionX, positionY}})
90         (Just TargetCapture {position = Robot {positionX = positionX', positionY = positionY'}}) =
91           sqrt ((positionX − positionX')^2 + (positionY − positionY')^2) ⩾ minDistanceChange
92       distanceChanged _ _                                                              = True
93        -- pomožna funkcija, ki glede na konstruktor posodobi vhodne podatke
94       squash = foldr squash' -- pregibamo od najstarejšega do najnovejšega
95         where squash' new@TargetCapture {} (t, _) = (t, Just new)
96               squash' new@TargetPosition {} (_, c) = (Just new, c)
```

```
97 {- Vstopna funkcija v program, ki sestavi mrežo. -}
98 main :: IO ()
99 main = do
100   prepareEnvironment
101   incubate $ do
102     -- operacija za upravljanje robota; nastavimo, da robot po ukazu za obračanje koles počaka, da se premik zaključi
103     nerveNXT     ← (growNeuron :: NerveOnlyFor NXTNeuron)      (λo. o {waitAfterSteering = True})
104     -- operacija za zajem položaja robota
105     nerveCapture ← (growNeuron :: NerveOnlyFrom CaptureNeuron) defaultOptions
106     -- operacija za prikaz položaja robota in pridobivanje lokacije tarče s strani uporabnika
107     nerveX11     ← (growNeuron :: NerveBoth X11Neuron)         (λo. o {showReachability = Just goodEnough})
108     -- operacija, ki usmerja robota k tarči
109     nerveTarget  ← (growNeuron :: NerveBoth TargetNeuron)      defaultOptions
110     -- podatke o položaju robota pošiljamo v prikaz in za usmerjanje k tarči
111     nerveCapture `attachTo` [TranslatableFor nerveX11, TranslatableFor nerveTarget]
112     -- podatke o lokaciji tarče pošiljamo operaciji, za usmerjanje k tarči
113     nerveX11     `attachTo` [TranslatableFor nerveTarget]
114     -- s podatki o potrebnih premikih robota za usmerjanje k tarči krmilimo robota
115     nerveTarget  `attachTo` [TranslatableFor nerveNXT]
116 {- ========== Od tu dalje sledijo razne pomožne definicije. ========== -}
117 {- Podatkovni tip stanja operacije. -}
118 data TargetState = TargetState {
119     target :: Maybe TargetForImpulse, -- zadnja znana lokacija tarče
120     capture :: Maybe TargetForImpulse -- zadnji znan položaj robota
121   }
122 type Target = StateT TargetState IO
123 {- Podatkovni tip operacije. Ne vsebuje podatkov sam po sebi. -}
124 data TargetNeuron deriving (Typeable)
```

```
{- Podatkovni tip podatka, ki izstopa iz operacije in nosi visokonivojske podatke usmerjanja robota. -}
data TargetFromImpulse = Move {
      impulseFromTimestamp :: ImpulseTime,
      forwardsOrBackwards :: Bool, -- smer premika
      steerTo :: Int, -- velikost zavoja
      speed :: Int -- hitrost premika
    } |
    Stop {
      impulseFromTimestamp :: ImpulseTime
    } deriving (Eq, Ord, Read, Show, Typeable, Data)
{- Podatkovni tip podatka, ki vzstopa v operacijo in nosi ali zajet položaj robota ali lokacijo tarče. -}
data TargetForImpulse = TargetCapture {
      impulseForTimestamp :: ImpulseTime,
      position :: Robot, -- položaj robota
      obstacles :: [Obstacle] -- položaji morebitnih ovir v poligonu, trenutno se ne uporablja
    } |
    TargetPosition {
      impulseForTimestamp :: ImpulseTime,
      targetX :: Double, -- X koordinata tarče
      targetY :: Double -- Y koordinata tarče
    } deriving (Eq, Ord, Read, Show, Typeable, Data)
type TargetOptions = NeuronOptions TargetNeuron
instance Impulse TargetFromImpulse where
    impulseTime Move {impulseFromTimestamp} = impulseFromTimestamp
    impulseTime Stop  {impulseFromTimestamp} = impulseFromTimestamp
    impulseValue Move {forwardsOrBackwards, steerTo, speed} =
      [toRational . fromEnum $ forwardsOrBackwards, toRational steerTo, toRational speed]
    impulseValue Stop  {}                                    = []
```

```
153 instance Impulse TargetForImpulse where
154   impulseTime TargetCapture {impulseForTimestamp} = impulseForTimestamp
155   impulseTime TargetPosition {impulseForTimestamp} = impulseForTimestamp
156   impulseValue TargetCapture {position = Robot {positionX, positionY, direction}, obstacles} =
157     [toRational positionX, toRational positionY, toRational direction] ++ obstacles'
158       where obstacles' = concatMap (λObstacle {obstacleX, obstacleY, diameter}.
159                                     [toRational obstacleX, toRational obstacleY, toRational diameter]) obstacles
160   impulseValue TargetPosition {targetX, targetY}                                    =
161     [toRational targetX, toRational targetY]

162 instance Neuron TargetNeuron where
163   type NeuronFromImpulse TargetNeuron = TargetFromImpulse
164   type NeuronForImpulse TargetNeuron  = TargetForImpulse
165   data NeuronOptions TargetNeuron -- ni nastavitev

166    -- začetno stanje je prazno, ne vsebuje trenutne tarče niti trenutnega položaja robota
167   live nerve _ = evalStateT (run nerve) (TargetState Nothing Nothing)

168 {- Prevajanje med zajetim položajem robota v položaj za usmerjanje robota. -}
169 instance ImpulseTranslator CaptureFromImpulse TargetForImpulse where
170   translate CaptureDescription {impulseTimestamp, robot, obstacles} =
171     [TargetCapture {impulseForTimestamp = impulseTimestamp, position = robot, obstacles}]

172 {- Prevajanje med lokacijo tarče s strani uporabnika v lokacijo za usmerjanje robota. -}
173 instance ImpulseTranslator X11FromImpulse TargetForImpulse where
174   translate X11Target {impulseFromTimestamp, targetX, targetY} =
175     [TargetPosition {impulseForTimestamp = impulseFromTimestamp, targetX, targetY}]
176   translate X11Keys   {}                                        = [] -- ignoriramo tipke

177 {- Prevajanje med visokonivojskimi podatki usmerjanja robota k tarči v nižjenivojske podatke za upravljanje robota. -}
178 instance ImpulseTranslator TargetFromImpulse NXTForImpulse where
179   translate Stop  {impulseFromTimestamp = impulseTimestamp}                                =
180     [NXTSteer {impulseTimestamp, tachoCount = 0}, NXTSpeed {impulseTimestamp, outputPower = 0}]
```

```
translate Move {impulseFromTimestamp = impulseTimestamp, forwardsOrBackwards, steerTo, speed} =
  -- vrstni red je pomemben, saj želimo najprej obrniti kolesa in se šele potem premakniti
  [NXTSteer {impulseTimestamp, tachoCount = steerTo'}, m]
    where m | forwardsOrBackwards = NXTSpeed {impulseTimestamp, outputPower = speed'}
            | otherwise           = NXTSpeed {impulseTimestamp, outputPower = -speed'}
          steerTo'                = max (-maxSteer) ∘ min maxSteer $ fromIntegral steerTo
          speed'                  = max (maxSpeed `div` 4) ∘ min maxSpeed $ speed
```